Faculteit der Exacte Wetenschappen

Master thesis

A Scalable RDF Store Based on HBase

Sever Fundătureanu

Supervisor **Paul Groth**
Second Reader **Jacopo Urbani**

Septemper 2012

Abstract

The exponential growth of the Semantic Web leads to a need for a scalable storage solution for RDF data. In this project, we design a quad store based on HBase, a NoSQL database which has proven to scale out to thousands of nodes. We adopt an Id-based schema and argue why it enables a good trade-off between loading and retrieval performance. We devise a novel bulk loading technique based on HBase coprocessors and we compare it to a traditional Map-Reduce technique. The evaluation shows that our technique does not scale as well as the traditional approach. Instead, with Map-Reduce, we achieve a loading throughput of 32152 quads/second on a cluster of 13 nodes. For retrieval, we obtain a peak throughput of 56447 quads/second.

Contents

1 Introduction **5**
- 1.1 Motivation . . . 5
- 1.2 The Approach Summarized . . . 8
- 1.3 Contributions and Outline . . . 8

2 Background **10**
- 2.1 Linked Data . . . 10
- 2.2 RDF . . . 10
- 2.3 HBase . . . 12
  - 2.3.1 Data model . . . 12
  - 2.3.2 Architecture . . . 13
  - 2.3.3 Hadoop integration . . . 15
  - 2.3.4 Advanced features . . . 15

3 Schema Design **16**
- 3.1 Access patterns . . . 16
- 3.2 Preliminary evaluation . . . 17
- 3.3 The Id-based Schema Overview . . . 18
- 3.4 General Design Questions . . . 19
- 3.5 Addressing the Research Questions . . . 22

4 Bulk Loading Techniques **23**
- 4.1 Map-Reduce Bulk Loading . . . 23
- 4.2 Coprocessors Bulk Loading . . . 27
- 4.3 Comparison between the 2 loading techniques . . . 28

5 Evaluation **30**
- 5.1 Experimental setup . . . 30
- 5.2 Bulk Load Performance . . . 30
  - 5.2.1 M/R Bulk Load Performance Break-Down . . . 32
  - 5.2.2 Coprocessors Bulk Load Performance Break-Down . . . 34
- 5.3 Retrieval performance . . . 37
  - 5.3.1 Overall Retrieval Performance . . . 38
  - 5.3.2 Retrieval performance break-down . . . 39

5.3.3 HDFS locality impact 42
5.3.4 Overall caching impact 43
5.3.5 Numerical data impact 44
5.4 Disk usage 45

6 Related Work **48**

7 Future work **52**

8 Conclusions **54**

A Evaluation Tables **55**

Chapter 1

Introduction

1.1 Motivation

The Semantic Web is “a common framework that allows data to be shared and reused across application, enterprise, and community boundaries” [26]. It leverages the current infrastructure provided by the World Wide Web (WWW) to provide scalable and ubiquitous access.

At the heart of the Semantic Web lies a set of principles for building Linked Data [49], a term coined by Tim-Berners Lee. Briefly, the principles are: use HTTP URIs to uniquely identify concepts, not just Web documents; use the Resource Description Framework (RDF) as the single data model and RDF links to build a graph of concepts.

RDF is the framework that enables making data re-usable, discoverable, easily accessible and easy to integrate from multiple and heterogeneous sources [33]. Unlike HTML which is used to make untyped connections between documents in the WWW, RDF enables typed statements between concepts in the world. Its main characteristics include a simple data model, formal semantics which enable inference, an extensible URI-based vocabulary and flexibility for representing data using different schemata. At the fundamental level, the data model is a collection of triples, each consisting of subject, predicate and object resources.

The Web is currently experiencing an explosion of RDF data. The Linking Open Data (LOD) [16] project is a community effort to convert open datasets into RDF format, make RDF connections between them and finally, make them publicly available. The latest statistics show that a cache of this data has reached an impressive 52 billion triples [16], with datasets that span multiple domains such as geographic, government, media, life sciences etc.

The exponential growth trend of the Semantic Web leads to a need for a scalable storage solution for RDF data. One storage option is to use Relational Databases which have been the mainstream storage solution for several decades. Although they have good performance and efficiency for many workloads, they can only scale to a certain extent at an expensive hardware cost [14][32].

As an alternative, NoSQL systems which try to solve the scalability limitations of Relational Databases have emerged. They are successfully deployed at scale in major internet companies such as Google [38], Facebook [29] and Amazon [20]. The main characteristics of NoSQL include being open-source, distributed, horizontally scalable and trading full ACID properties for performance [19].

The broad aim of this project is to investigate what is the most efficient way of loading RDF data into a NoSQL database, while keeping good retrieval performance. There are a wide range of available NoSQL systems, so we performed an analysis to determine which one best suits our requirements for RDF storage.

Four main NoSQL categories can be delimited:

- Key Value/Tuple Stores: simple data model, associates a unique key with a particular item of data; e.g. Voldemort [30]
- Wide Column/Column Family Stores: similar to key value stores, but data can be stored in multiple columns which are grouped in column families; e.g. Google's BigTable [38] and its open-source version HBase [32], Facebook's Cassandra [2]
- Document Stores: store semi-structured documents which are a collection of fields and attachments serialized in formats like JSON; e.g CouchDB [3]
- Graph Databases: store data according to graph structure: nodes, edges, and properties; e.g. Neo4j [18]

In order to make the most appropriate choice, we need to reason further on the requirements for storing RDF data. For each triple, we want to track provenance, as this can have multiple metadata functions such as the source of the document containing the assertion, the author, confidence, date etc. The triple store has to have enough flexibility to define the context either at the statement level or at a more coarse-grained level through named graphs. Furthermore, we want the store to provide enough flexibility to map its schema on RDF schemata at any level, from the fundamental assertions to complex ontologies. Finally, we also want to use our RDF data in deductive reasoning algorithms. This means that our distributed store should be easy to integrate with a distributed computational framework that enables inference.

A more concrete requirement is to provide support for efficiently answering numerical SPARQL range predicates. This requirement comes from the context of the

OpenPhacts [21] project, which aims at building a hub of integrated biological and chemical data. Its main research question is to find “all oxidoreductase inhibitors <100nM active in both human and mouse”. Such questions effectively translate into SPARQL range-predicates which need to be answered in an efficient manner.

From the above requirements, Key-Value stores and Document Stores do not match very well with our use case, mainly because they lack in flexibility. The schema associated to Key-Value stores is too simple and would lead to unnecessary workarounds for storing more complex RDF schemata. Similarly, Document Stores are not a natural fit with RDF data because they have been designed to be document-centric as opposed to data-centric. They typically use verbose serialization formats such as JSON, which have an impact at large scale on network and disk usage.

Graph databases might appear to be a good match in terms of schema design. Indeed, through RDF statements the data effectively becomes linked into a global graph. However this project aims at using a proven solution that can scale out to hundreds and even thousands of nodes while still maintaining good retrieval performance. Typically, graph database solutions do not perform very well at such scale, one example being Neo4j [17] .

Finally, we choose HBase [32] because it provides considerable flexibility in schema design, it is open-source and it is becoming a mature project with a sustained support from the developer community. It has been successfully used in large-scale production systems such as Facebook’s messaging system, reaching thousands of nodes [29]. HBase provides strong consistency, which guarantees that reads will never return stale data. This is the reason we favored it over Cassandra, which provides only eventual consistency. Furthermore data availability is insured through replication, which is a built-in feature of the underlying Hadoop Filesystem (HDFS) [6]. Finally, integration with Hadoop makes it a perfect candidate for running large-scale map reduce inference algorithms [56] in an efficient manner.

Thus, the concrete aim of this project is to investigate how we can use HBase to efficiently store large-scale RDF data. The research is going to focus on two main questions:

1. How should we design the HBase *schema* such that it enables the best bulk loading performance, while maintaining good retrieval performance and efficiently satisfying numerical range-predicates?

2. Which *techniques* provided by HBase enable the best performance for *bulk loading* RDF data?

1.2 The Approach Summarized

The schema we employed is based on the optimized index structure presented by Harth et al. [46]. We associate each non-numerical triple atom to an 8-byte id using 2 *Dictionary* tables, *Value2Id* and *Id2Value*. We then use these ids to build 6 *Index* tables that cover all possible access paterns when querying combinations of subject(s), predicate(p), object(o) and context(c). For each *Index* table, in non-object positions, we store 8-byte ids, while in object positions we additionally store numerical literals which are encoded as their corresponding internal Java representation.

Although [46] mentions this schema is suited for B+ trees, *HBase* can also leverage it successfully because it sorts table keys lexicographically and allows querying based on key prefix and key ranges. From all of *HBase*'s operations, *Range Scans* offer the best read throughput because they make use of data locality. Furthermore this schema makes efficient use of disk space as it only stores the verbose triple atoms in the Dictionary tables, while the *Index* tables with higher cardinality have a more compact representation. A disadvantage of this schema is the overhead added upon loading for generating unique ids for each non-numerical element. Managing these ids during incremental bulk loads can also become problematic.

The first technique for bulk loading data into this schema is based on Map-Reduce [11] and uses a total of 10 jobs. 2 main stages can be identified:

- Id Generation: we use 2 Map Reduce jobs to generate the id-based quads in SPOC order plus the mapping between ids and non-numericals
- Loading HBase Tables: for each table we run a Map Reduce job that will sort the rows in lexicographic key order and generate Hadoop *Sequence Files* in internal *HBase* format; then, the files are validated and directly adopted by *HBase*; for this stage, we need 2 jobs for *Dictionary* tables and 6 jobs for generating each *Index* table.

A second technique for bulk loading RDF data is based on *HBase* coprocessors [9]. Coprocessors offer the possibility of running user code on the servers making up the cluster. Among other functions, they can be used to efficiently build secondary indexes. Our intention is to bulk load one of the 6 index tables e.g. SPOC and then use the coprocessors to generate all the other indexes.

We will evaluate an id-based schema with 2 loading techniques and build a position for which one can best satisfy the research questions.

1.3 Contributions and Outline

The contributions of this thesis are as follows:

- applying and tuning the schema presented in [46] for HBase
- evaluation of a novel bulk loading method based on HBase coprocessors
- analyis of retrieval performance using our tuned schema
- multiple ideas to extend the current solution into a full fledged triple store

In chapter 2 we introduce basic concepts related to Linked Data, RDF and HBase. Chapter 3 argues our choice for an HBase schema. We then present in more detail the 2 loading techniques in chapter 4. We evaluate the store both in terms of bulk loading and simple retrieval operations in chapter 5. Chapter 6 compares our work with other triple stores and chapter 7 presents ideas for future work.

Chapter 2

Background

In this chapter, we start by introducing the basic concepts of Linked Data and RDF. Then, we present HBase, providing a more detailed view because the reader needs a good understanding of this complex system in order to understand our analysis in subsequent chapters.

2.1 Linked Data

Linked Data is a set of guidelines for exposing, sharing and connecting pieces of information on the Web [33]. It is based on 4 main principles:

- using URIs to uniquely identify concepts and things in the world, not just Web documents
- using HTTP dereferencing to have a concept URI point to a Web document that provides a description of that concept
- using RDF as the standard data model to structure information
- using RDF links to make connections between concepts in different datasets

When publishing Linked Data on the Web the following steps need to be taken [36]: assign URIs to entities and provide associated URIs for their description set RDF links to existing data sources on the Web provide metadata in order for clients to be able to evaluate the quality of the published data

The final result, the Web of Data, can be described more accurately as a "web of things in the world, described by data on the Web"[36].

2.2 RDF

The Resource Description Framework (RDF) is a generic datamodel which standardizes the way in which linked data should be structured. The information is effectively built into a graph of data and each connection in this graph is equivalent to an RDF *statement* or *triple*. A *triple* consists of a subject(s), a predicate(p) and an object(o). The subject and object represent nodes in the graph, while the predicate represents the edge. Additionally, for tracking provenance, each *statement* is endowed with a context (c) element. In the latter case, we use the term *quad* to refer to a *statement*.

The *subject* is either an URI that points to a resource or a blank node. Blank nodes are nodes without a URI reference. The *predicate* can only be a URI as it denotes a typed link between the subject and the object. The *object* is either an URI, a blank node or a literal. Literals are nodes with no outgoing edges. They can be plain strings or typed literals i.e. numbers, dates etc. In the provenance mode, the *context* can also be either a URI or a blank node.

For example, a statement can specify that the city of Cambridge has an urban population of 130000 people. This triple is represented in the Turtle serialization format as follows:

```
<http://dbpedia.org/page/Cambridge>
<http://dbpedia.org/ontology/populationUrban>
"130000"^^<xsd:integer>.
```

The population value is represented through a typed numerical literal with the type xsd:integer.

For querying RDF, we will make use of *statement patterns*. For the more general case of *quads*, such a pattern has the form `(S,P,O,C)`, where S,P,O,C are either variables or constants (also known as bound variables) in Subject, Predicate, Object or Context positions respectively. The variables will be represented with '?' characters. As an example, a query asking for all the statements which share the same subject will have the form `(S,?,?,?)`.

At the fundamental level RDF presents a very abstract model. In order for different datasets to use the same domain-specific terms, more concrete models have been developed [33]. These are taxonomies, vocabularies and ontologies expressed in SKOS (Simple Knowledge Organization System) [58], RDFS (the RDF Vocabulary Description Language, also known as RDF Schema) [40] and OWL (the Web Ontology Language) [23].

Finally, datasets can appear in multiple RDF serialization formats:

- RDF/XML - maps RDF triples on the XML structure
- RDFa - embeds RDF triples in HTML documents
- Turtle - plain text with support for prefixes and other shorthands

- NTriples - similar to Turtle, but more redundant; enables one line per triple parsing
- RDF/JSON - an effort to map RDF triples on top of JSON (not yet standardized)

2.3 HBase

HBase, the open-source twin of Google's *BigTable* [38], is a wide column NoSQL solution that trades relational features for performance at the cost of denormalizing tables. The same built-in mechanisms used for providing horizontal scalability are also used for providing fault tolerance and data availability [32] .

In section 2.3.1, we will describe the logical structure of a table, together with the basic operations provided by HBase. In section 2.3.2 we will explain how the logical structure maps on the physical architecture, and how the data flows through the system during basic operations. Section 2.3.3 will explain how HBase integrates with Hadoop. Finally, section 2.3.4 will present some of HBase's advanced features which are going to be used in this project.

2.3.1 Data model

The basic logical unit in HBase is a *column*. Several columns compose a *row*, which is uniquely addressed by a row key. A set of rows composes a *table* and all rows within a table are sorted lexicographically. For each row and for each column we have a multi-version cell. A cell version is a timestamped snapshot of a certain cell value. This effectively adds an extra dimension for addressing the data within a table.

Columns are grouped into *column families*. The important distinction is that column families have to be specified at schema design time, while columns can be dynamically added. Both the number of rows and columns in a table is unlimited from a logical standpoint. Conversely, the number of *column families* should be kept to a small number because it can cause needless I/O [12].

A full column name is referenced as *family:qualifier*. Both the row key and the column qualifier are an arbitrary array of bytes, not necessarily human-readable.

The four basic operations allowed in HBase are: Put, Get, Scan, Delete. A *Put* operation inserts or updates a single row in a table. If the row key already exists, the cell value is updated with the new one. Similarly a *Get* operation retrieves a single row and a *Delete* operation removes a single row. The API for issuing these operations is simple and intuitive:

```
void put(Put put) throws IOException
Result get(Get get) throws IOException
void delete(Delete delete) throws IOException
```

Each of these operations have wrapper objects passed as parameters. These objects are created by specifying a row key in their constructor. Afterwards one can optionally call *addFamily()* or *addColumn()* to narrow down the scope of the operation to a specific column family or column.

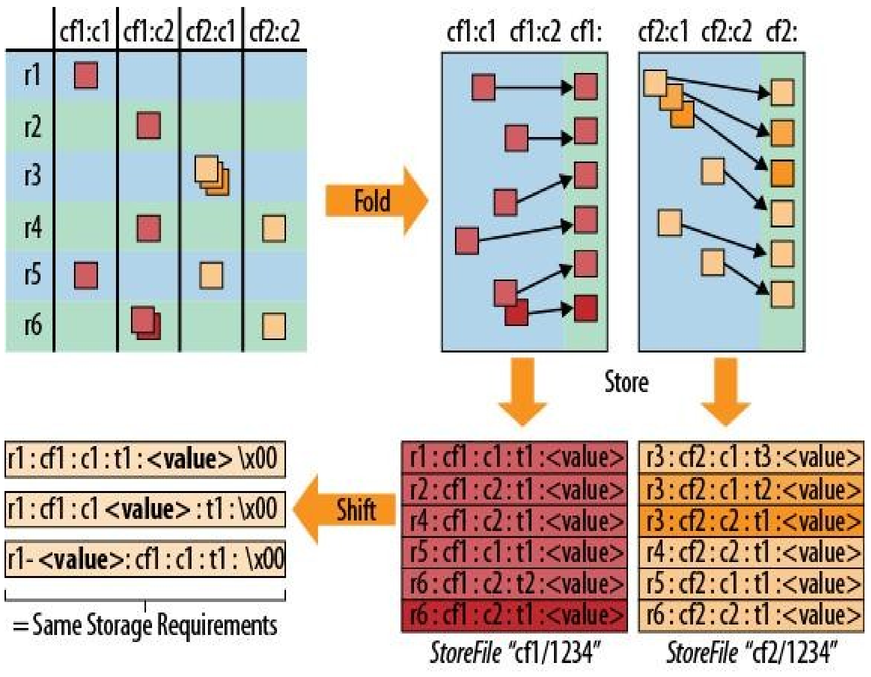
A *Scan* operation can be used as an iterator starting with a given row. An associated Scan object is created, again, using a row key in its constructor. The iterator is obtained through the call

```
ResultScanner getScanner(Scan scan) throws IOException
```

and from here on the client can use *next()* to walk through the rows.

2.3.2 Architecture

Each table in *HBase* is horizontally split into *Regions* which are distributed to *Region Servers* (RS). Only one *RS* owns a certain Region at one point, so any request for a certain key is always routed through the same region server. Thus strong consistency is insured.



cf1:c1
cf1:c2
cf2:c1
cf2:c2
r1
r2
r3
r4
r5
r6
Fold
cf1:c1 cf1:c2 cf1:
cf2:c1 cf2:c2 cf2:
Store
r1 : cf1 : c1 : t1 : <value> \x00
r1 : cf1 : c1 <value> : t1 : \x00
r1- <value>: cf1 : c1 : t1 : \x00
= Same Storage Requirements
Shift
r1 : cf1 : c1 : t1 :<value>
r2 : cf1 : c2 : t1 :<value>
r4 : cf1 : c2 : t1 :<value>
r5 : cf1 : c1 : t1 :<value>
r6 : cf1 : c2 : t2 :<value>
r6 : cf1 : c2 : t1 :<value>
r3 : cf2 : c1 : t3 :<value>
r3 : cf2 : c1 : t2 :<value>
r3 : cf2 : c2 : t1 :<value>
r4 : cf2 : c2 : t1 :<value>
r5 : cf2 : c1 : t1 :<value>
r6 : cf2 : c2 : t1 :<value>
StoreFile "cf1/1234"
StoreFile "cf2/1234"

Figure 2.1: Translation from logical to physical data model. Lars George. *HBase: The Definitive Guide*. OReilly Media, 2011.

In figure 2.1, we present the translation from the logical data model to the physical one. Each *column family* within a region translates into separate HDFS directories called *Stores*. Each *Store* is made up of multiple on-disk *StoreFiles* and, we also add, an in-memory *Memstore* - not shown in the picture. Each entry in these files is a key-value pair in which the key is composed of <logical rowKey,column family, column qualifier, timestamp> and the value contains the associated logical cell value. The Memstore is an unsorted buffer, while the StoreFiles are sorted ascending by logical row key, column family, column qualifier and descending by timestamp. The bottom left part of the figure shows how the logical data can be shifted between row key, column qualifier and cell value, while having almost the same physical representation on disk. This is important to keep in mind when designing the schema.

For each *Put* operation the write path is as follows: *Zookeeper* [44], a distributed coordination service which HBase depends upon, redirects the client to the *RS* responsible for that key; the request is first appended to a log used as backup for the *Memstore* ; the physical key-value pair is then appended to the *Memstore* ; when the *Memstore* is full, the pairs are sorted and flushed to disk into a new *StoreFile*. The important observation here is that the same *logical row key* can become part of multiple *Store Files* because the *physical row key* also contains the column qualifier.

For *Get* and *Scan* operations, the read path starts with a similar redirection to the appropriate *RS*. This in turn will start a *merged-read* operation, which means that the requested row key will be searched in the *Memstore* and all *StoreFiles*. A *StoreFile* is divided into blocks which are indexed by the first row key in a *block index* residing at the beginning of each *Storefile*. Thus a lookup in a *Storefile* implies a search in the block index, followed by a sequential search in the loaded block. This process can be improved by enabling Bloom Filters. This lookup exposes an important feature of HBase which is *cost transparency*: if a *Store* has 5 *Storefiles*, the user knows that at most 5 disk seeks will be made in the read operation.

In order to maintain a good read performance, HBase also makes use of *compaction* processes [32] to merge files in a *Store* and keep their number low. There are 2 types of *compactions*: a *Major Compaction* compacts all the *Stores* in a table and after it is run it leaves one *StoreFile* per *Store* - it is scheduled once a day by default; and a *Minor Compaction* picks up a couple of small files in a *Store* and merges them - it is triggered automatically depending on certain server-side parameters.

Fault tolerance is handled at the filesystem level through replication. So if a *Region Server* crashes, each of the other servers will pickup a part of its regions directly from HDFS.

2.3.3 Hadoop integration

Both Hadoop and HBase use the same underlying filesystem HDFS. This means that the user can get the best of both worlds: he can run batch jobs using Hadoop to achieve maximum throughput and use HBase to provide a good latency over the same data. HBase's internal files are nothing more than special HDFS Sequence Files.

An important use case for this integration is bulk loading large volumes of data. HBase provides the API to properly configure a Hadoop job based on the number of currently deployed regions in a table. The outputed files from the Hadoop job can be adopted directly by the Region Servers after a proper validation.

2.3.4 Advanced features

Filters are an important feature which give the user more fine-grained control over the data that he wants to include in the query results. The benefit is that they run on the server-side, so if your load is well enough distributed, the entire cluster can be used. Examples include filtering rows based on key prefixes or some regular expression. Filters can also be combined in a *FilterList*, from which all or only one needs to pass.

Coprocessors provide even more flexibility than filters. They are essentially user-code that can be deployed in the HBase cluster. Mainly, there are 2 types of coprocessors:

- Observers - for each base operation i.e. Put, the user can deploy interception hooks that can either pre-process the input parameters or post-process the results
- Endpoints - act like stored procedures and can be invoked through Remote Procedure Calls (RPC)

In this chapter, we introduced Linked Data and RDF. Then we provided a detailed view of HBase, with a stronger accent on its data model and underlying architecture. For further details about this complex system we refer the reader to Lars George's comprehensive definitive guide [32].

Chapter 3

Schema Design

The design of an HBase schema is a complex process driven by a particular use-case in the architectural constraints of HBase. In section 3.1 we introduce the notion of *access patterns* as an important driver for the design process. Next, we make a preliminary evaluation of these *access patterns* in section 3.2. In section 3.3 we provide an overview of the Id-based schema. We then explain in section 3.4 how we chose its composing elements by answering some general design questions. Finally, in section 3.5 we show how this schema addresses our research questions.

Our use-case consists of bulk loading RDF data as efficiently as possible into a data store and then provide fast and flexible retrieval capabilities over that data. The schema has to provide support for datasets containing triples with metadata (*context*) - *quads*, but we also want to track datasets containing simple triples. A more specific use case will be to efficiently store and retrieve quads/triples containing numerical literals.

3.1 Access patterns

The first step in designing the schema is realizing what is the *data access pattern* after being loaded into HBase. From the aforementioned use-case this access pattern is *random reads*. For example, taking the query `(S,?,?,?)`, we want to jump to the location in the dataset where all the information associated to `S` resides.

We now have to match the *data access pattern* on a set of HBase *table access patterns*. We first need to understand what *table access patterns* can be employed and what are their strengths and weaknesses. There are 2 possible cases:

1. Random reads - do not perform very well because the *merged reads* are expensive especially when there is a large number of *Storefiles* ; for each *Storefile* the needed block is first located in the block index, then the key is looked up sequentially in the loaded block

2. Sequential reads - perform better than (1) because they do only one lookup for the start key and then read all the required keys sequentially from there on

3.2 Preliminary evaluation

Our goal is to store the data in a way that maximizes read throughput. Therefor, we did a preliminary evaluation of the access patterns offered by an HBase table. The results contributed to the design of our schema. The evaluation was made

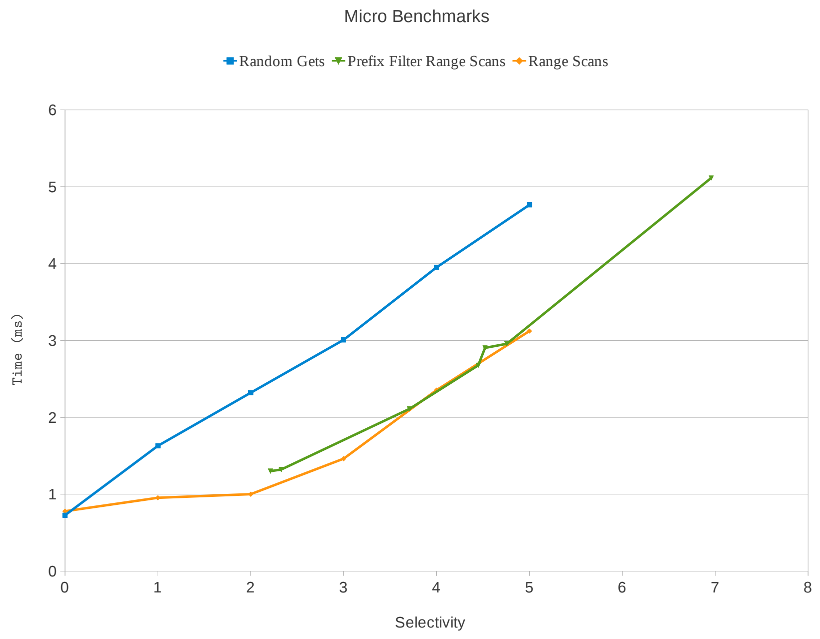


Figure 3.1: Micro Benchmarks

on a 32 node cluster, running Dual CPU Dual-Core (4 cores) AMD Opteron(tm) 280 Processors@2.4GHz with 4GB RAM. We loaded a dataset of 10 million 24-byte sequential keys in a table with 1 column family, 1 column qualifier and empty cells. We want to understand how each operation scales with the number of queried keys, so we vary *selectivity* in powers of 10 from 1 to 100000. We ran the following micro benchmarks:

- Random Gets: we run *selectivity* Get operations in which the keys are randomly chosen
- Range Scans: we run one Scan operation in which the start key is randomly selected and the end key is computed as *startKey+selectivity* ; scanner caching is set to the minimum value between selectivity and 500; this value represents the number of rows fetched in one *next* call

- Prefix Filter Range Scans : similar to range scans, except that the Scan operation is defined through a start row key and PrefixFilter

The graphs in figure 3.1 are presented in log-log base 10 scale. They show that Range Scans can perform even one order of magnitude better than Random Gets. The Prefix Filter Range Scan performs as good as the normal Range Scan. Random Gets scale linearly with selectivity, but Range Scans show a smaller slope until a selectivity 100, meaning they perform even better than linear for that range. The results clearly show that Range Scans i.e. sequential reads should be used as much as possible. Thus, an important goal to keep in mind when desiging the schema is that we have to try to convert the random-reads *data access pattern* to a sequential-reads *table access pattern*.

3.3 The Id-based Schema Overview

In this section we present a brief overview of the schema, while in section 3.4 we give a more detailed argumentation for our choices. The Id-based schema - figure 3.2 - makes use of the optimized index structure presented by Harth et al. [46] and David Wood [61]. The similarities with their work are as follows:

- we associate each non-numerical triple atom to an 8-byte id using 2 Dictionary tables *Value2Id* and *Id2Value*

- with these ids we build 6 Index tables that cover all possible access paterns when querying combinations of subject(s), predicate(p), object(o) and context(c).

The distinction we make is for the special case of numerical literals which are stored in O-positions. These are encoded as their corresponding internal Java representation, so they do not need any id mapping. We distinguish between numericals and non-numericals using an additional *type* byte to prefix O-positions. Thus each row key in the index tables has a total of $8 * 3 + 9 = 33$ bytes, while the column qualifiers and cell values are left empty. The column family name is chosen as short as possible 'F', because it appears in all physical key-value pairs.

For the Dictionary tables, we use one column family 'F' as well, but we also add one column (blank name) in which we store the mapped data. In the *Id2Value* table we map the 8-byte ids to serialized Sesame values [27], while in the *Value2Id* table we map MD5 hashes of Sesame values to the 8-byte ids.

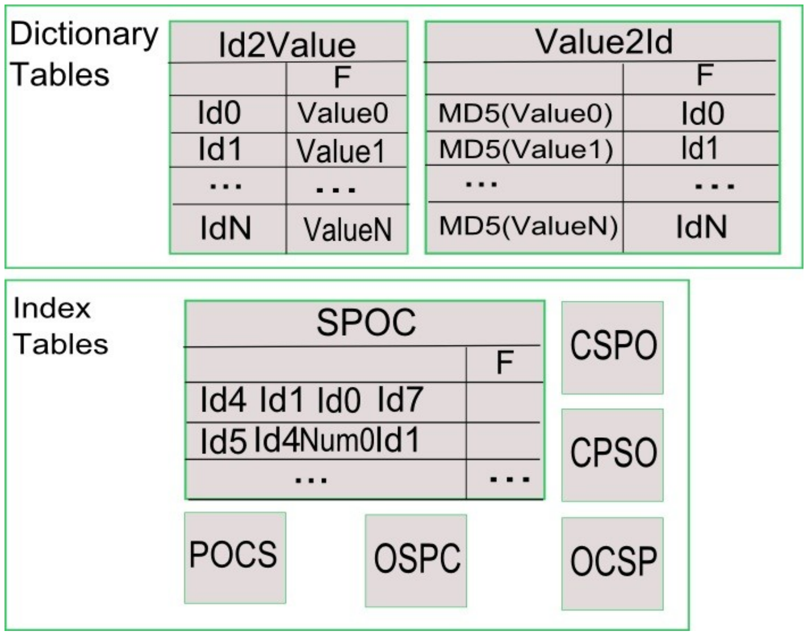


Figure 3.2: HBase Id-based Schema

3.4 General Design Questions

In general, when designing an HBase schema the following questions need to be answered [13]:

1. How is the data seggregated into *tables* and *column families*?

In our case, the data is seggregated into *tables* based on access pattern. We saw that sequential reads give the best performance, so we place the bulk of the data, which is now compressed to 33 bytes/quad, into Index tables. The 2 Dictionary tables usually have a reduced cardinality compared to the Index tables, because many resources appear in multiple statements since they act as nodes in a graph. Given this reduced cardinality we consider acceptable for these tables to offer the random reads *table access pattern.* Still, we will employ further optimizations such as caching to take into account performance penalties. For all tables, one *column family* is enough for meeting our requirements.

2. How are *row keys* and *columns* chosen? Do we opt for a *tall-narrow* or a *flat-wide* design?

The most important thing to realize in row key and columns design is that you can move your data from the cell value to the column qualifier or even the row key, because it translates in a physical key-value pair with the same information, only a bit shifted. For example, if we store the value 'v' in the column 'cf:c' with row key 'k' it translates into the physical representation: ['k','cf','c',timestamp,'v']. If we move the value into the row key the physical representation becomes: ['kv', 'cf','c',timestamp,00]. The advantage of the latter example is that we can do prefix matching i.e. we can request the same entry using either 'k' or 'kv' as row key. This is possible because HBase stores pairs lexicographically sorted by row key.

We make use of prefix-matching to reduce the number of necessary Index tables to cover all possible query patterns. We shift all ids making up the quads into the row key, so that we are able to cover as many query patterns as possible with only one table. For example, the SPOC index table will be able to answer the query patterns: `(S,?,?,?)`, `(S,P,?,?)`, `(S,P,O,?)` and `(S,P,O,C)`. The other 5 Index tables are POCS, OCSP, CSPO, CPSO and OSPC. This effectively leads to a *tall-narrow* design.

The *flat-wide* alternative would have been to have 15 index tables to cover all combinations of s,p,o,c. For example, the table S-P-OC would have stored in one row all the quads associated to a certain subject. The subject would have been used as a row key, the predicate as a column qualifier and the objects and contexts concatenated in the cell value. One disadvantage of this schema is the increased disk requirements to replicate all the id-based quads 15 ways. If we multiply that by the HDFS replication factor, usually 3, it becomes obvious that this solution hinders scalability.

Furthermore, the *flat-wide* design affects load balance granularity. In order to manage growing data, HBase has to split regions and maybe move them around the cluster for load balancing. This region split can only be made at row granularity level. So if a single row reaches millions of colums, all this information will be forced to reside in one region. In a *flat-wide* design, for a table storing information by predicate i.e. P-O-CS, this is likely to take place. Conversely, in our *tall-narrow* design the tables are easily splittable, so a more fine-grained load balance is insured.

An advantage of *flat-wide* tables are that you can make use of the row-level atomicity to modify multiple columns of a single row atomically. However that feature does not apply to our use case, since we are only interested in reading the data after bulk loading it into the store.

For the Dictionary tables, we use one column in which we map the desired row key. For the *Value2Id* table we need to keep in mind that row keys can not be very large because they become part of the block index in which the lookups should be as fast as possible. Actually, HBase currently limits the size of a row key to 32768 bytes. As a result, in the *Value2Id* row key we use the 16-byte MD5 hash of the mapped element. As will be shown in section 5, this will not affect query performance because only the bound variables will be retrieved from this table, so maximum 4 *Gets* per query. Furthermore the hash will randomly distribute the row keys to Regions, which enables good load balance during bulk loading.

In the *Id2Value* table, we map each id to a serialization of an RDF element. Each cell value starts with 1 *type* byte: 2 bits indicate the main type - URI, Literal or BNode; for Literals other 2 bits specify if its a Plain, Datatype or Language literal. Following is the element content in the case of URIs and BNodes, or a list of tokens in the case of Literals - {label} or {label,language} or {label,datatype}. The label is prefixed by 4 bytes length while the other tokens (datatype or language) by 2 bytes length.

3. Which important HBase configurations need to be considered?

HBase offers many optimizations through configurations that can be applied so that a particular schema performs as best possible. The one we consider the most important is deciding the caching strategy. As we will show in the evaluation chapter, the most overhead during queries is shown while mapping ids back to their corresponding string values. As a result we decided to cache only the *Id2Value* table, pointing out that for the Index tables the probability of cache hits is small. Because of its reduced cardinality, and supposing there is enough RAM allocated to each *RegionServer*, the *Id2Value* table can ideally fit entirely into cache which will give a significant retrieval performance improvement.

For each table, we also need to consider the block size. When looking up a key in a *Storefile* it is first searched in the block index, then the required block is loaded into memory and the key is searched sequentially within the block. When doing random reads this process is repeated over and over. Harth et al. [47] have shown that the lookup process can be improved by using smaller blocks. Although this will increase the block index size, it will shorten the sequential search within the block. As a result we use 8K blocks for the tables *Id2Value* and *Value2Id*, while for the index tables we consider the default of 64K blocks to be appropriate for sequential reads.

For improving read performance, we also enabled Bloom filters on all tables. This will significantly reduce the number of disk seeks when there are many *Storefiles* in a *Store*. Moreover, since a lookup also involves a sequential search within a block, this optimization will prevent loading of unnecessary blocks into memory. On the other hand, these Bloom filters are also loaded into cache so we have to manage their size.

Another important configuration option to consider is compression. This can only be applied on cell values, so we enabled it on the *Id2Value* table. Since we store a lot of text in this table, this option will significantly reduce both network and disk I/O.

3.5 Addressing the Research Questions

The Id-based schema favors good loading performance because it maintains a compact representation of the bulk of the data. As will be shown in the evaluation section, the disk requirements for this schema are approximately the same as the input data in format NQuads, even with our 6x replication.

Good retrieval performance is expected because the bulk of the entire operation is effectively translated into a Range Scan and only the unique ids have to be mapped through Random Gets back to their associated strings. The latter step can be further optimized through caching and appropriate choice of block size.

Numerical range-predicates are also efficiently satisfied because they translate into a Range-Scan. For example the question ”find potency between 0.001 and 0.01” can be resolved as follows:

- compute a start key that consists of the 8-byte id representing the predicate ”potency” concatenated with the 8-byte internal Java representation of 0.001
- compute an end key in a similar manner but with the 8-byte internal Java representation of 0.01
- using these keys, run a range scan in the POCS table and retrieve all context-subject ids of interest
- map ids back to their string representation

In order for the numerical ordering to coincide with the lexicographic ordering we flip the sign bit in our representation, so that negative values are lower than positive ones.

In this chapter we presented the reasoning behind designing our HBase schema. We introduced the notion of *access patterns* as an important driver for the design process. Then, we made a preliminary evaluation of these *access patterns* and concluded that *Sequential Reads* should be used as much as possible. Next, we provided an overview of the Id-based schema and explained how we chose its composing elements by answering some general HBase design questions. Finally, we pointed out how the schema addresses our research questions.

Chapter 4

Bulk Loading Techniques

In this chapter we present the 2 bulk loading techniques we employed in this project. Both of them use Map-Reduce(M/R), a distributed framework introduced by Google, designed for efficiently processing huge datasets on a large number of computers [41]. While the first technique is solely based on M/R jobs, the second one uses HBase coprocessors [9] to generate 5 of the 6 index tables.

4.1 Map-Reduce Bulk Loading

The pure Map-Reduce Bulk Loading [11] technique leverages the fact that HBase's internal *StoreFiles* are a special case of Hadoop *SequentialFiles*. As a result they can be generated using Hadoop jobs and then adopted directly by HBase through an API call. The technique has also been evaluated by Barbuzzi et. al [35] and showed to perform better than a normal insertion technique from the client side.

The entire process totaling 10 Map-Reduce jobs is presented in figure 4.1. The diagram shows how the input and output is connected between jobs. We also mark the order in which the jobs are being run. Overall, 2 main stages can be identified:

- Id generation: the verbose triples in *NQuads* format are converted into an unsorted dictionary and a list of id-based quads in SPOC order.
- Loading HBase tables: for each table we run a Map Reduce job that will sort the rows in lexicographic key order; for this stage we need 2 jobs for the Dictionary tables and 6 jobs for generating each Index table.

Id Generation

For the Id generation stage, we considered 2 possible techniques. One is inspired from Urbani's work [55], which uses 2 Hadoop jobs but introduces gaps in the id-space. Another technique [5] uses 3 Hadoop jobs, but generates consecutive ids with no gaps. We favor loading performance over the gaps in the id-space and choose

Urbani's solution with a slight modification with the goal of minimizing the unused id-space.

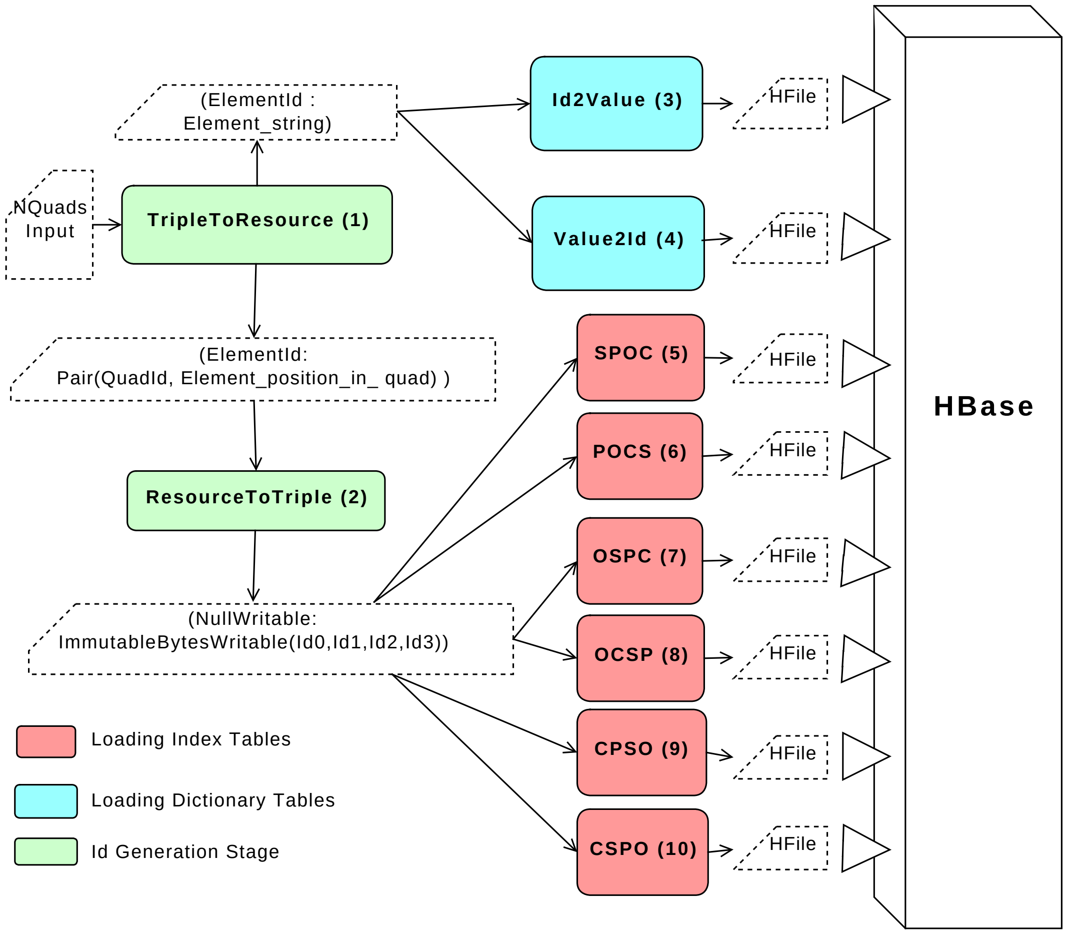


Figure 4.1: M/R Bulk Loading Technique

We present the process briefly and we refer the reader to [55] for a more detailed view. The dictionary encoding can be summarized as follows:

- *TripleToResource* job: breaks the quads into their composing elements and generates unique 8-byte ids for each element:
  - mapper: generates an 8-byte id for each quad from 5 bytes partition id concatenated with 3 bytes local counter; emits the pair `(Element_string, Pair(QuadId, Element_position_in_quad))`
  - reducer: For each quad elemenent it generates an *ElementId* in the same manner: 5 bytes partition id + 3 bytes local counter. Differently, [55] uses 3 bytes partition id + 4 bytes local counter. However, we consider this division to introduce too large gaps because one reducer has to process $2^{32} = 4.29$ billion elements to fully utilize the local counter. Another task

of the reducer is to write the mappings `(ElementId, Element_string)` to Hadoop *Task Side-Effect Files* [7] and then emit the pair `(ElementId, Pair(QuadId, Element_position_in_quad))`

- *ResourceToTriple* job: rebuilds the quads using the ids generated in the previous job:
    - mapper: reads output from previous job and swaps ElementId with QuadId emitting `(QuadId, Pair(ElementId, Element_position_in_quad))`
    - reducer: concatenates the element ids into a byte array in SPOC order and wraps it into an *ImmutableBytesWritable* object which is the format used internally by HBase

Loading HBase Tables

For bulk-loading an HBase table through Map-reduce, 2 steps have to be followed:

1. Run a Hadoop job which generates HBase *Storefiles* in a separate HDFS directory. We had to implement only the mapper. An API call automatically configures the reducer to emit the output format expected by HBase and the partitioner to impose total order on the row keys.
2. Adopt the generated files into HBase through another API call. This adoption is an HDFS move operation if HBase uses the same underlying HDFS cluster as the one in which the files have been generated. If the HDFS clusters are different, the adoption process becomes a copy operation, so it is more expensive.

For loading the tables *Value2Id* and *Id2Value*, we ran a Hadoop job which reads the *Side-Effect Files* generated in the reduce phase of the *TripleToResource* job of the Id generation stage. For the *Value2Id* job, in the mapper, we generate the MD5 hash from each quad element and emit `(ImmutableBytesWritable(hash), Put(hash, id))`.

The Hadoop jobs for loading the Index tables read the id-based quads outputed from the *ResourceToTriple* job of the Id Generation stage. The mappers simply re-order the quad ids accordingly and emmit the pair `(ImmutableBytesWritable(id-based_quad), Put(id-based_quad, NULL))`.

For each Index table loading job, the number of reducers is configured automatically to be the number of regions in that table. Upon creation any table has only one region, so for efficient loading we need to pre-split the tables. The split is essentially dictated by the element acting as prefix in the table's key. Thus, we have the following situations:

- The prefix is a non-object position e.g. SPOC : for these tables the splits are determined from the range of generated ids; specifically, we track the last generated id and we divide the id-space into 2*cluster_size equally sized regions; this number is chosen so that we use all available reducer slots (2 per node).

- The prefix is an object position e.g. OSPC : for these tables, we need to take into account that O-positions are prefixed by a type byte which has the first bit 1 for numerical literals and 0 for the rest; thus a certain number of regions fall in the range [0x80..00-0xff..ff] while the rest make up the space in [0-Last_generated_id]; for allocating a proportional number of regions to each range, we count numerical literals during the *TripleToResource* job.

Shuffle Optimization

The Shuffle stage is the part of a Map-reduce job during which the output from the mappers is partitioned, sorted and transfered to the reducers. For a more detailed description of the process, we refer the reader to [31].

On the mapper side, for each task, one file has to be produced which is then going to be fetched by the reducers. This file can be produced either from a single disk spill, either from multiple spills followed by successive merges. Obviously the first case means less disk I/O, so that is the goal of our optimization.

Each mapper has a circular buffer in which the output data is accumulated. When the buffer reaches a certain threshold a background process starts to spill the buffer to disk. Importantly, if the spill does not finish before the buffer fills up entirely, the mapper is blocked. This leads to 2 alternatives for our configuration: first, if we can make a good estimation of our task output, we configure the circular buffer size to fit the entire mapper output and we set the spill threshold to the maximum value (1.0) so that we spill only when the task is finished; second, if we can only make a rough estimation of our task output or if the process memory does not allow for a circular buffer too large, we do not want to risk blocking the mapper, so we set the spill threshold to a lower value i.e. 0.8.

Generally, in the second category fall the jobs which deal with the string values of RDF elements. The size of these strings is variable so we can only make a rough estimate of the output of these mappers. Conversely, for jobs which deal only with 8-byte ids a more accurate estimation can be made for the output of each mapper, so we can better optimize them if there is enough process memory for the circular buffer.

4.2 Coprocessors Bulk Loading

Coprocessors are hooks that run directly within the HBase region server processes. Therefore, they make a very good alternative to running M/R jobs on top of the HBase cluster. One of their applications, as pointed out by the HBase community, is generating secondary indexes [25]. In our schema, we can also consider to have a main table i.e. SPOC, while the other 5 index tables act as secondary indexes, because they store the same information only in a different order.

We make the following changes to the pipeline of jobs presented in the pure M/R technique:

- the *Id Generation* stage is merged with the loading of the SPOC table: in the reducer of the *ResourceToTriple* job, we use HBase Put operations to insert data into the SPOC table. This will ensure that HDFS data locality is achieved, which is important because the coprocessors will read data from the SPOC table.
- We replace the 5 jobs for generating the other Index tables with an RPC call to endpoint coprocessors deployed on each Region Server. The coprocessors will in turn generate the other 5 tables by scanning in parallel the SPOC table and inserting data into the other 5 using normal Put operations. Figure 4.2 demonstrates a subset of this process: loading the POCS table on 3 Region Servers.

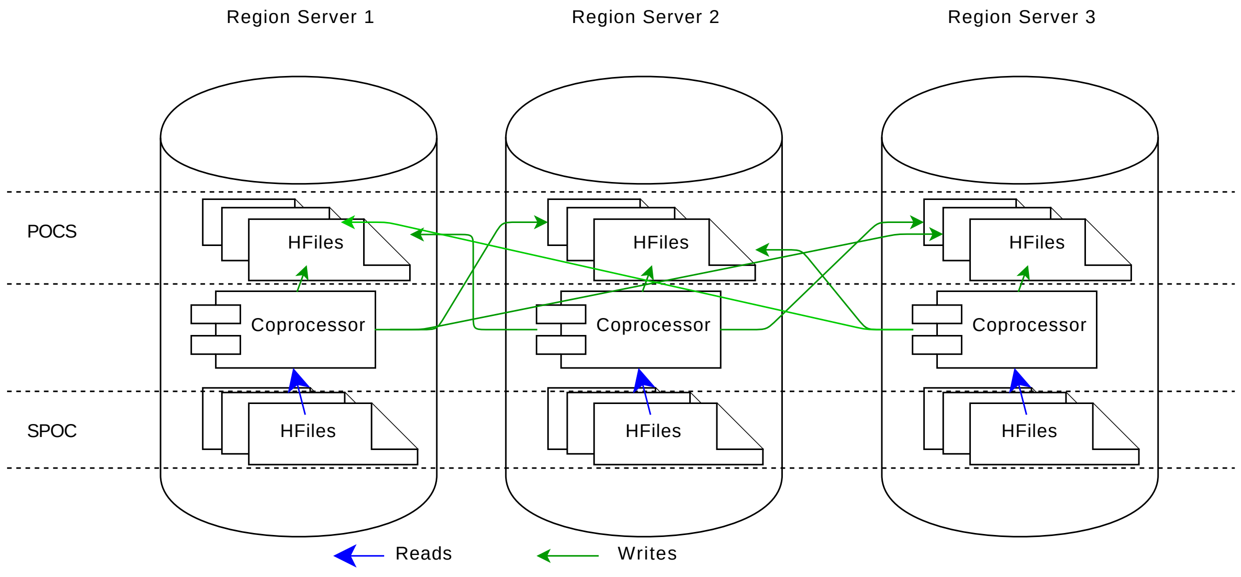


Figure 4.2: Coprocessors Loading Technique

We point out that our implementation can not be considered eventually consistent, because we use blocking RPC calls to all coprocessors, which leave the system in a consistent state when they finished running. Also, we maintained a lightweight implementation with the purpose of evaluating scalability, keeping in mind to add functionality for fault tolerance if results are good enough. However, chapter 5 will prove this is not the case.

Write Optimizations

The *Coprocessors* use HBase's normal write path: the client buffers a configurable number of Put operations, then it splits the buffer into batches per *RegionServer* and issues the batches in parallel; on the *RegionServer* the writes are accumulated in a *Memstore*, which is flushed to disk into a new *Storefile* when it gets filled. When the number of *Storefiles* in *Store* hits a certain limit, *compactions* will be triggered to merge and reduce the number of these files.

One optimization was to increase the client's buffer size to 40MB in order to batch as many Put operations as possible. The performance gain comes from a reduction in the number of RPC calls made between the coprocessors and the *RegionServers*.

Another optimization was to decrease the number of *compactions* that take place during loading because they consume both CPU and memory. In this sense, we increased the number of Storefiles that trigger a *compaction* (hbase.hstore.compactionThreshold) to the value of 10, and the maximum number of files that can be compacted at once (hbase.hstore.compaction.max) to 15.

A problem that had to be prevented was blocking writes [22]. The flushing of the *Memstore* would sometimes block because a maximum number of *Storefiles* would be reached, and *compactions* would not reduce their number quick enough. Configuration parameters were set accordingly to avoid this problem.

4.3 Comparison between the 2 loading techniques

A first difference between the 2 loading techniques is in terms of load balancing. While for the pure M/R technique load balancing is dictated by the number of mappers and reducers in each job, the Coprocessors version goes through HBase's online API, so it benefits from automatic load balancing.

Regarding the state the system arrives at, there is a difference in HDFS data locality for the 2 techniques. For the pure M/R technique, the HDFS Datanodes actually hosting the data are not necessarily colocated with HBase's RegionServers responsible for managing that data. This is because the data is generated outside the RegionServers and adopted afterwards through an HDFS move operation, which only changes the path of the files on the Namenode without physically moving the data. Conversely, the Coprocessors version goes through HBase's online API, meaning that all requests go through the Region Servers which will prefer local, rack-local and remote-rack Datanodes in this order. This effectively achieves full data locality.

Another difference is in the number of Storefiles created with each technique, which is important because it impacts the retrieval process. For the pure M/R technique, Storefiles are the output files of reducers, so one Storefile per Region is created. With the Coprocessors version we use HBase's normal write path, so many Storefiles will be created since the Memstore has a limited size (160MB per Region) leading to multiple flush operations.

In this chapter we presented the 2 bulk loading techniques we employed in this project. The first technique is based only on M/R jobs, while the second one uses HBase coprocessors to generate 5 of the 6 index tables. For each of them, we also explained some of the optimizations applied. Finally, we pointed out several differences between the 2 techniques.

Chapter 5

Evaluation

5.1 Experimental setup

We evaluated the id-based schema on bulk loading performance with the 2 techniques presented in chapter 4 and retrieval performance of our datalayer API. The tests were made on 13+1 (head node) nodes of the DAS4 cluster hosted at Vrije Universiteit, Amsterdam. The slave nodes have Dual CPU Dual Core (4 cores) AMD Opteron 280 Processors@2.4GHz with 4GB RAM, while the head node has 24 cores Intel Xeon(R) X5650@2.67GHz with 48GB RAM. The 13 nodes are located on the same physical rack.

We used HBase 0.92.0 and Hadoop 1.0.0. HBase and Hadoop daemons ran on all 13+1 nodes making up the cluster, while Zookeeper is deployed on 3 of these nodes. For fairness reasons, we allocated 2 GB for the HBase daemons and 2 GB for the Hadoop+HDFS daemons. Each Region Server is configured with 12 handlers (worker threads). Hadoop runs with 2 slots per map and 2 per reduce. The chosen HDFS block size is 128MB as it is more suitable for large datasets. For monitoring various HBase parameters we have also deployed Ganglia [4] within our cluster.

5.2 Bulk Load Performance

In table 5.1, we present total loading times and throughput for quad datasets generated with *Berlin SPARQL Benchmark (BSBM)*. We varied the *product count* parameter for generating the datasets from 40k to 4 million, resulting in datasets of sizes 11.48 million, 113.41 million and 1.133 billion quads. We generated the datasets in format TriG and then converted them to NQuads using the tool *rdfconvert* [24]. Additionally, we loaded a real-world dataset, namely *DBPedia* which contains 385.84 million quads. Due to time constraints, we performed the experiments only once. However, we point out that the difference between successive runs can be in the order of several minutes at most, which for large datasets is insignificant.

Table 5.1: Loading Time and Throughput

Dataset size	Throughput (Quads/second)		Total Time(min)	
	M/R	Coprocessors	M/R	Coprocessors
11.48 mill	11,799.864	13,016.012	16.219	14.703
113.41 mill	32,152.150	29,692.900	58.790	63.659
1.133 bill	27,241.369	20,229.351	693.128	933.384
385.84 mill (DBPedia)	29,812.760	23,307.200	215.706	275.914

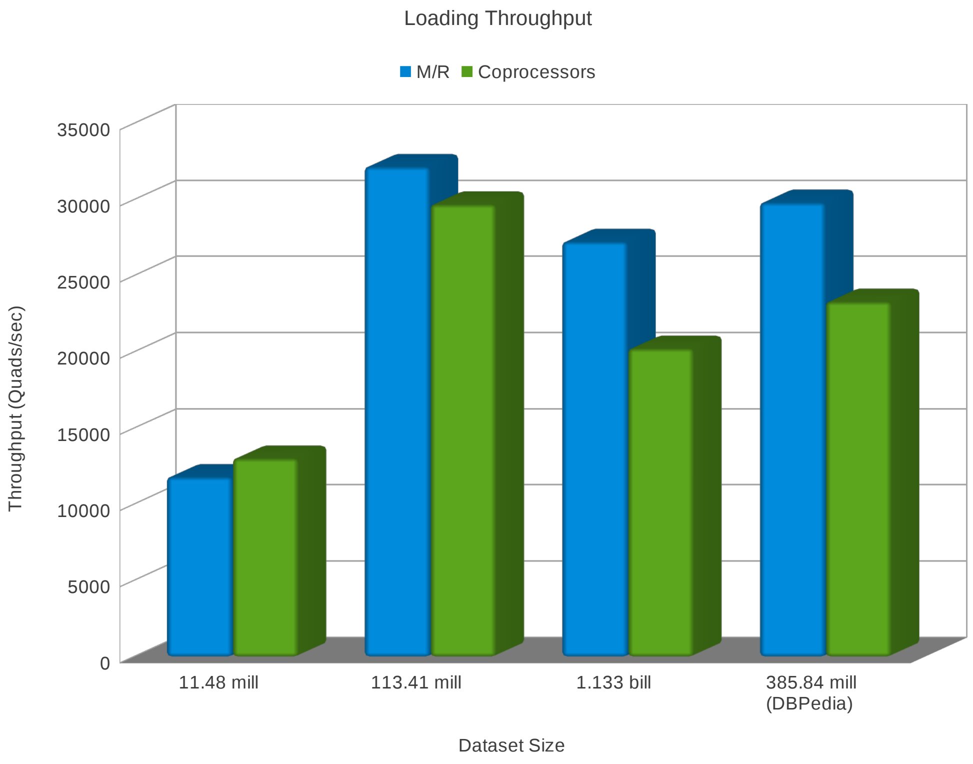


Figure 5.1: Overall Loading Throughput

Overall, we notice the Map-reduce bulk loading technique performs better than the Coprocessors version for the medium and large datasets. Although there are 10 Map-reduce jobs to perform with the pure M/R technique, the Coprocessors version exhibits significant overhead and does not scale very well. Still, for the smaller dataset, the Coprocessors version performs better because it does not have exhibit the startup overhead of a M/R job which can reach even 30 seconds.

For an in-depth understanding of the loading performance, we continue by presenting, in the subsections 5.2.1 and 5.2.2, the break-down per stage for each loading technique.

5.2.1 M/R Bulk Load Performance Break-Down

In figure 5.2, we present the performance break-down per stage expressed as percent of total loading time. We group the *IdGeneration* stage with loading the *SPOC* table in order to have a proper comparison with the Coprocessors version. Overall, we notice that the bulk of loading time is divided between *IdGen+SPOC* and the *Secondary Index* tables, the fractions varying between 40%-50%. On the other hand, the loading of *Dictionary* tables takes up a smaller portion of the total time with growing dataset size. This is expected because ids are reused in multiple quads, so the *Dictionary* tables remain small.

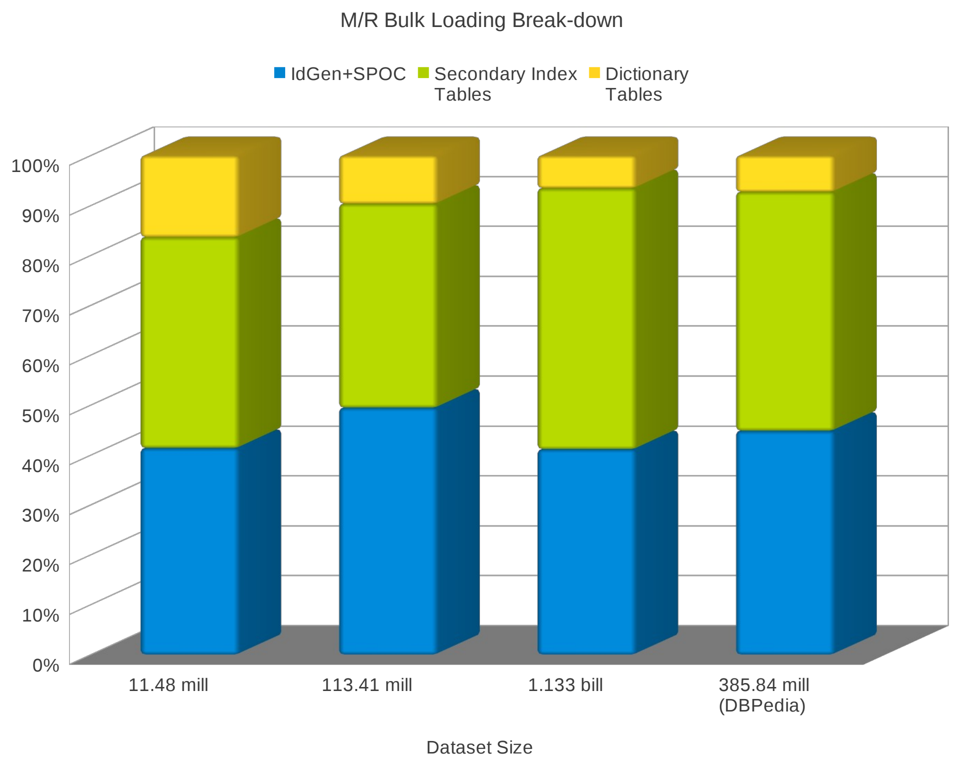


Figure 5.2: M/R Bulk Loading Breakdown

Since percentages are relative, we use figure 5.3 to understand how each stage scales. We use the loading throughput per stage computed as the input size of that stage divided by its running time. Since each stage has different throughput ranges, we normalize the measurements to the interval [0,1].

We notice the *IdGen+SPOC* and the *Dictionary* stages scale well, showing increased throughput with growing dataset size. Conversely, loading of Secondary Index tables reaches the peak throughput of 393,303 quads/sec for the 113 million dataset while for the 1.133 billion it drops to 260,401 quads/sec. To further understand we zoom-in on this stage and see how each table performs.

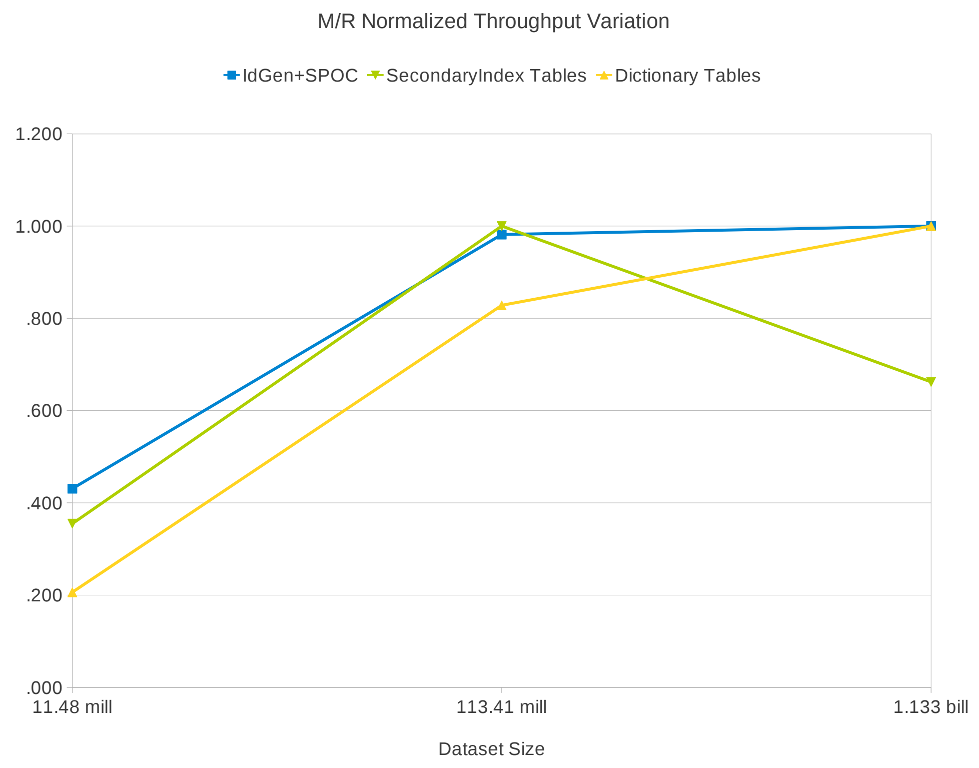


Figure 5.3: M/R Normalized Throughput Variation Per Stage

In figure 5.4, we show the variation of absolute throughput for each index table. We notice that all tables show the same behaviour of peaking for the 113 million dataset and dropping for the 1.13 billion one. For 3 of them the drop is steeper than the other 3, because they perform much better for the middle dataset reaching a peak of 516,946 quads/sec for the SPOC table. This behavior is the result of improper load balance. The chosen ranges are essential when the index tables are pre-split. Our simple division of the id space into equally sized ranges needs improvement so that 2 important factors are taken into consideration: data skew and the gaps left in the id-space in the *Id Generation* stage.

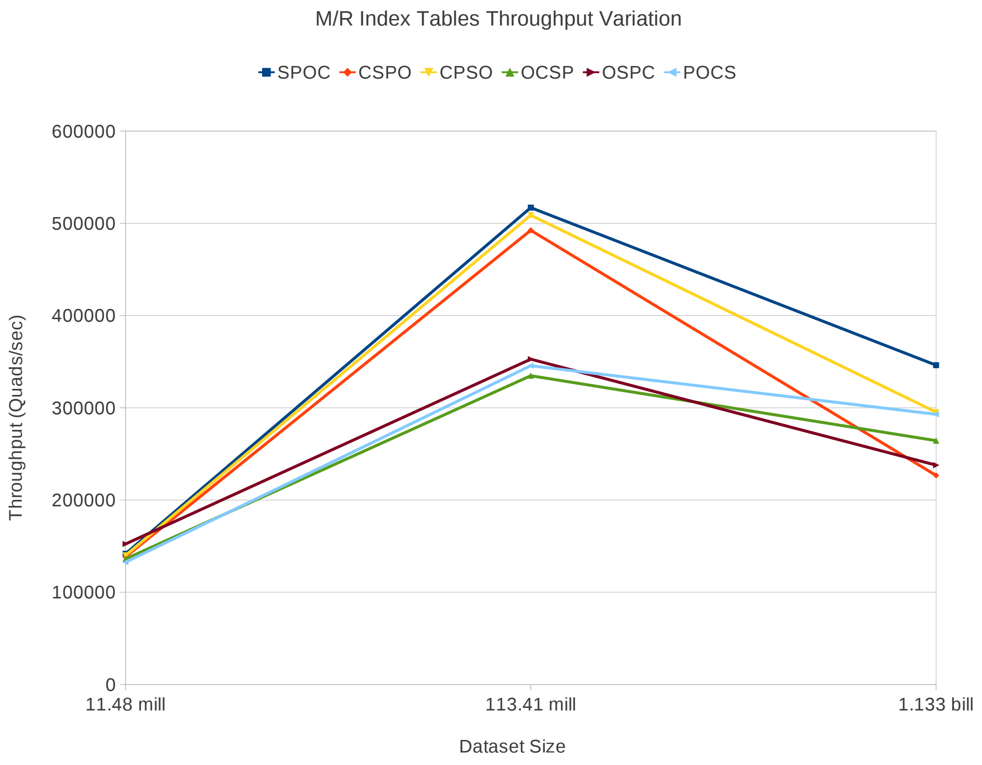


Figure 5.4: M/R Index Tables Throughput Variation

5.2.2 Coprocessors Bulk Load Performance Break-Down

Similarly, we look at the performance break-down for the Coprocessors bulk loading technique. In figure 5.5, we notice that the loading of Secondary Index tables, generated using coprocessors, takes up an increasing fraction of loading time with growing datasets. Looking at the throughput variation per stage in figure 5.6 we notice that the *Secondary Indexes* stage scales the worst. The *IdGen+SPOC* stage also exhibits a drop in throughput between the medium and largest dataset because of the online insertion operations used to load the SPOC table.

The performance drop exhibited by the Coprocessors comes also from HBase's internal operations. First of all, when multiple Put requests coming from different coprocessors, target the same region on a Region Server, an overhead will appear from the synchronized access to that region's *Memstore*. Second, for each *Memstore* flush a new *Storefile* is going to be created. When the number of *Storefiles* reaches the configured number of 10, HBase triggers the *compaction* for that *Store*, a process which consumes both CPU and memory. To confirm, we checked the compaction queue size exposed by each *RegionServer*. Figure 5.7 shows the variation for a subset of the *RegionServers* caught with Ganglia.

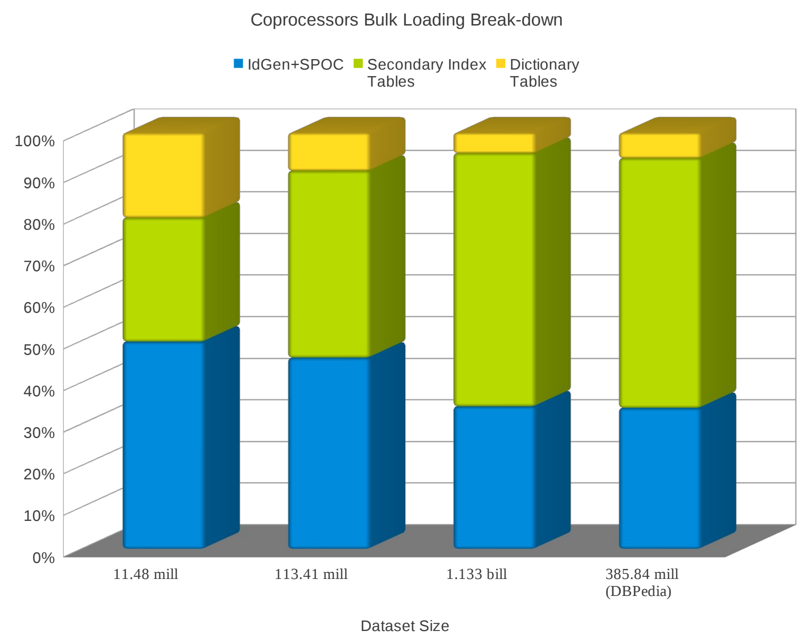


Figure 5.5: Coprocessors Bulk Loading Break-down

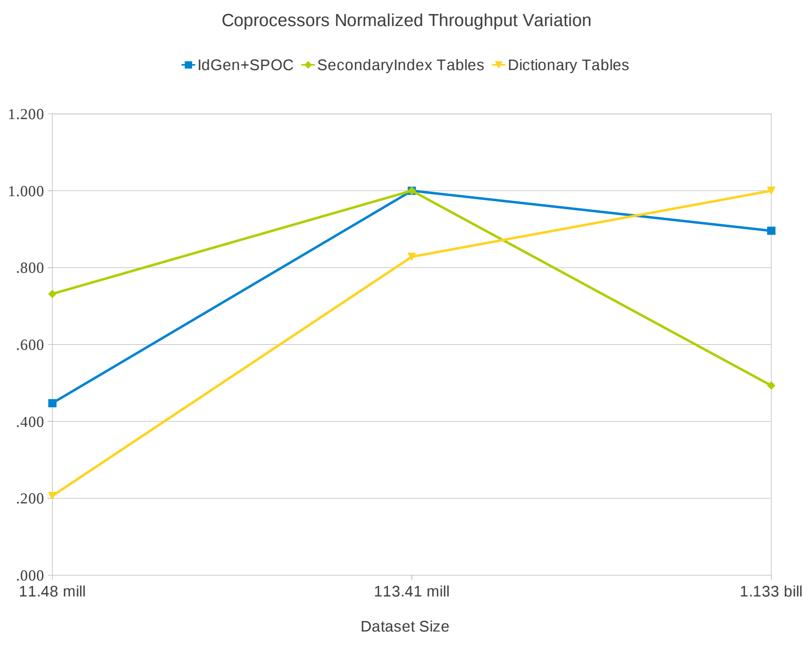


Figure 5.6: Coprocessors Normalized Throughput Variation Per Stage

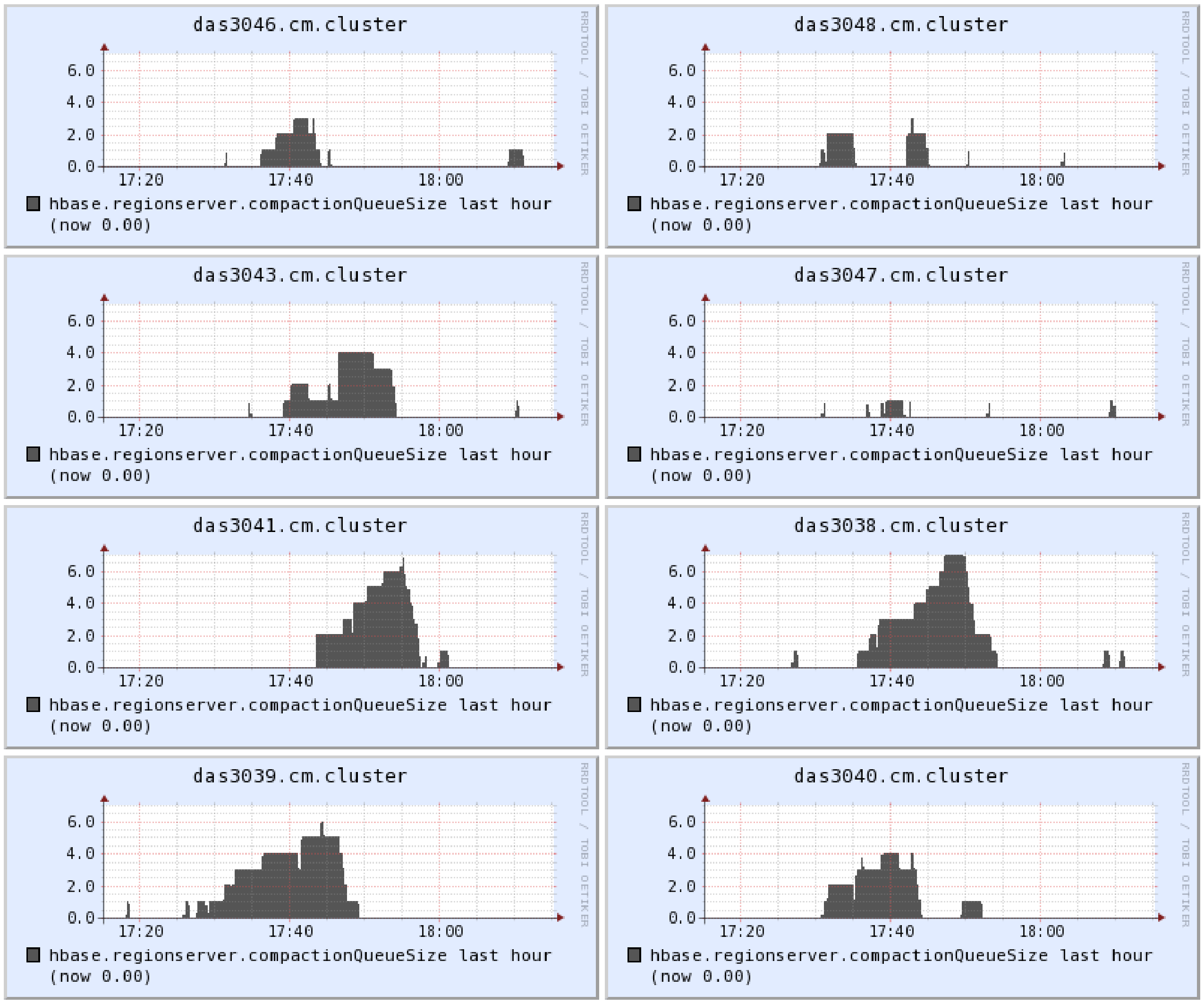


Figure 5.7: Compaction Queue Size during Secondary Index Generation

We also point out that, unlike the M/R technique, the *Secondary Indexes* stage for the Coprocessors version is well balanced, because the coprocessors use normal Put operations and HBase balances itself automatically. To confirm, we show the variation of write requests, for the DBPedia dataset, for a subset of HBase's Region Servers in figure 5.8. At approximately 16:40 the coprocessors start to run and we notice that the requests are well distributed across servers. However, we point out that maintaining good load balance also adds overhead since regions have to be split and moved around the cluster.

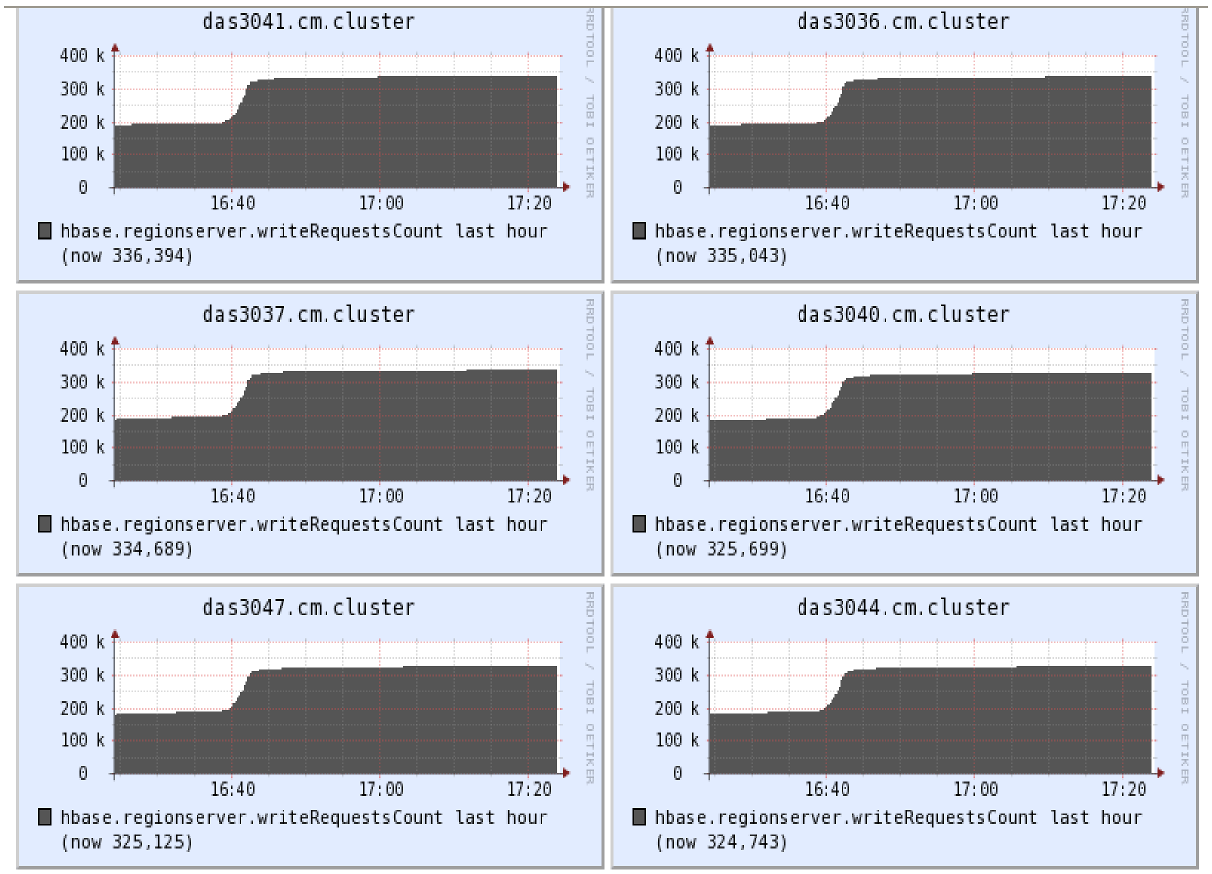


Figure 5.8: Balanced Cluster during Coprocessor Loading for DBPedia

5.3 Retrieval performance

In this section, we present measurements for the retrieval throughput of datasets mentioned in section 5.2. Since we did not implement support for SPARQL, we can not use the queries provided by already existing benchmarks. Instead, we have designed an ad-hoc benchmark for queries of simple complexity(without joins) which use directly the API provided by our datalayer. We designed the benchmark with the following requirements in mind:

- queries should present a wide-range of selectivities; this is important because the throughput has a significant variation depending on selectivity
- queries should cover all access patterns i.e. combinations of s-p-o-c, which ensures that all Index tables will take part in the benchmark

- for queries with the same selectivity the results are accumulated into a mean value

The strategy we employed is to start with a set of main queries which return a large number of results e.g. `(?, <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/producer>,?,?)`. For each of these queries we randomly choose 10 of its results and derive other queries so that all possible patterns are covered. For example, from the result quad `(s1,p1,o1,c1)` we derive the queries `(s1,?,?,?)`, `(s1,p1,?,?)`, `(?,?,o1,?)` etc. All the derived queries are gathered into a list, from where they are issued in a random order so that we are not biased towards caching results. As an optimization we set the number of results returned by the scanner in one *next* call depending on query pattern. The heuristics are inspired from [54]. For this benchmark, the throughput results will be presented as points in the graphs corresponding to particular selectivities. From these points, we extrapolate a logarithmic trend line to give an average quantification for a benchmark run.

In subsection 5.3.1, we show overall retrieval performance for each dataset, then we dive into the performance break-down in subsection 5.3.2. We continue with 3 subsections that show how retrieval performance is affected by HDFS locality 5.3.3, block cache size 5.3.4 and presence of numerical data in the result set 5.3.5.

5.3.1 Overall Retrieval Performance

We ran the benchmark for each dataset size mentioned in section 5.2 after doing a *Major Compaction*, because, as we will see in subsection 5.3.3, that is when the maximum read throughput is achieved.

In figure 5.9, we notice 2 general trends for the retrieval throughput, with a peak of 56447 quads/second. One is for the small and medium dataset the other is for DBPedia and the 1.1 billion BSBM dataset. One reason for the 2 separate trends is related to the caching of the Id2Value table. For the higher trend, the Id2Value table is fully cached, while for the smaller trend only part of this table can fit into the cache. This and the random access to this table results in high cache churn.

Although the DBPedia dataset is smaller than the 1.11 billion BSBM dataset, it performs about the same. We point out that the ratio `total_input_elements:unique_ids` is 2.0 for DBPedia and 3.6 for the large BSBM dataset. This means that the BSBM dataset reuses more ids than DBPedia. As a result the DBPedia queries have to map more ids back to their associated strings, thus the performance drop.

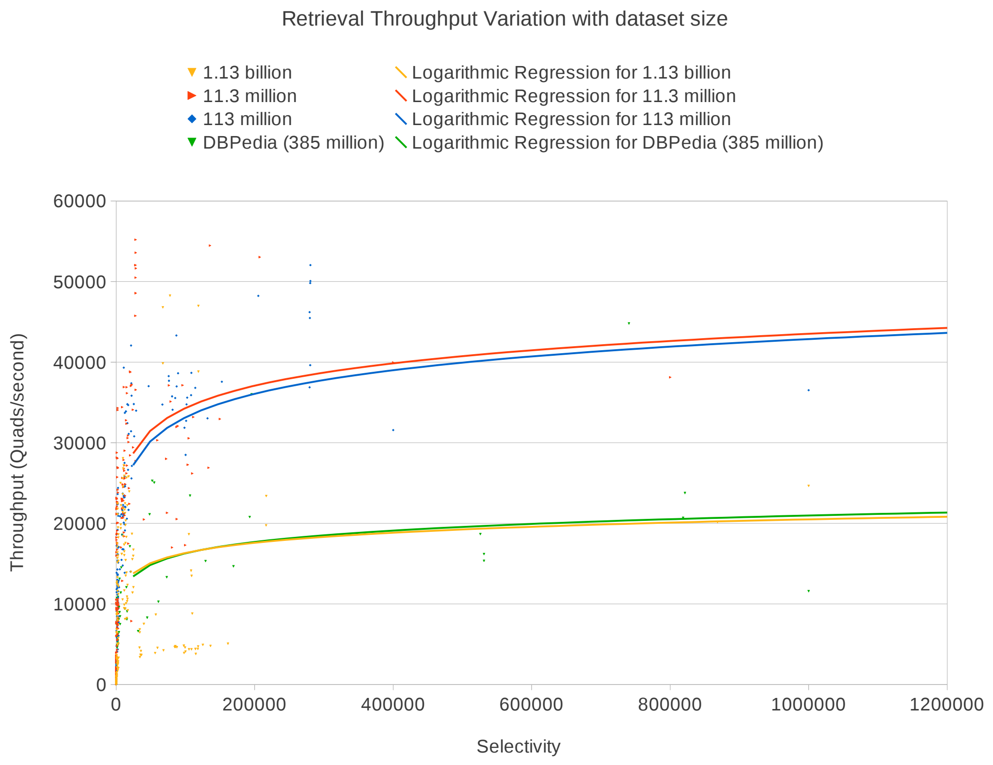


Figure 5.9: Retrieval Throughput Variation with Dataset Size

5.3.2 Retrieval performance break-down

There are 3 steps that compose a query:

- *Value2Id* - maps the bound variables to their associated ids
- *RangeScan* - does a range scan in the appropriate index table with the prefix built from the ids obtained in the previous step
- *Id2Value* - maps the ids from the results obtained in the previous step back to their corresponding *Value* instances

In this section we show how each of these steps make up the total retrieval time.

We choose a set of queries of small complexity and issue them on the DBPedia dataset to show this performance break-down. We consider 3 selectivity ranges: high [0-200], medium [1000-15000] and small - 1 million results. We ran the queries both with a cold and a warm cache in order to justify our caching configuration and show how caching affects the overhead distribution. Running with cold caches means we ran the benchmark immediately after bulk loading. The break-down for each case is expressed as percentages of retrieval time.

In figure 5.10, we present the break-down with cold caches. We notice that for all 3 selectivity ranges the *Id2Value* step takes up the most overhead. This confirms that random reads are indeed much less performant than range scans. We notice that for high selectivity, this step takes up a smaller fraction, which makes sense because these queries usually have more bound variables leaving a smaller number of unknown ids to be mapped back in the results.

The least overhead fraction is taken by the *Value2Id* step in all 3 cases. This makes sense because, although this step's access pattern is also random reads, it involves only the bound variables in the input query, so maximum 4 elements.

The *RangeScan* step takes up 32% for queries with high selectivy, while for medium and small it decreases to 14.13% and 13.7% respectively. The overhead fraction decreases with decreased selectivity, because range scans scale better than random reads.

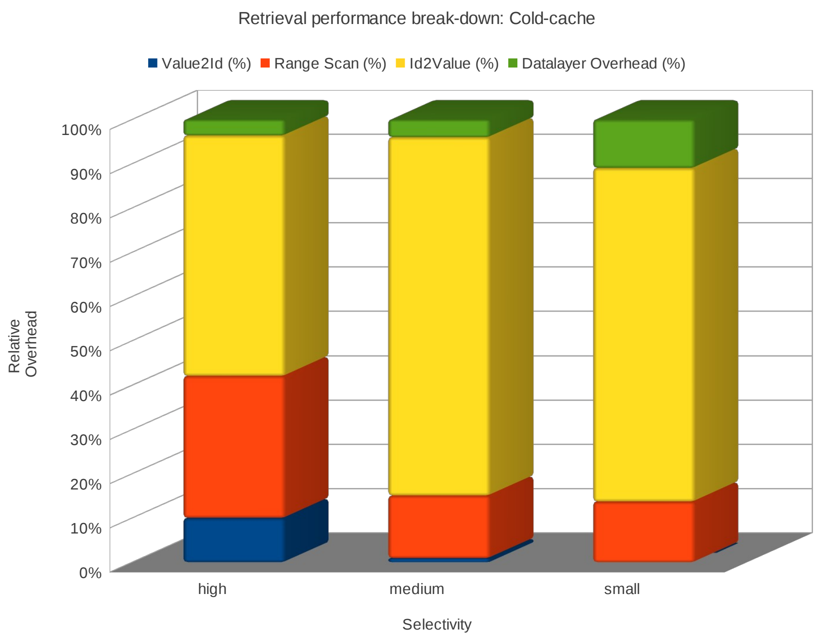


Figure 5.10: Retrieval Break-Down: Cold Cache

The datalayer overhead comes esentially from building the final results from the list of id-based quads retrieved from the *RangeScan* and the map built in the *Id2Value* step. This takes O(n) time, so for large result sets the overhead becomes noticeable.

From the above, it is obvious that the *Id2Value* step is the main bottleneck of the whole retrieval operation. So the optimization to cache this table is justified and even necessary for decent retrieval performance. This is further strengthened by the fact that the *Id2Value* table has a reduced cardinality compared to the index tables, so a large portion, ideally the whole table, can fit into the block cache. This will favor increased cache hits. On the other hand, the *Value2Id* step makes up an overhead fraction too small to justify including the corresponding table into the cache. Caching the index tables is also not justified, because as the data volume increases there is a small chance that consecutive queries will need to fetch the same rows from these tables. Thus, the only table for which caching has been enabled is *Id2Value*.

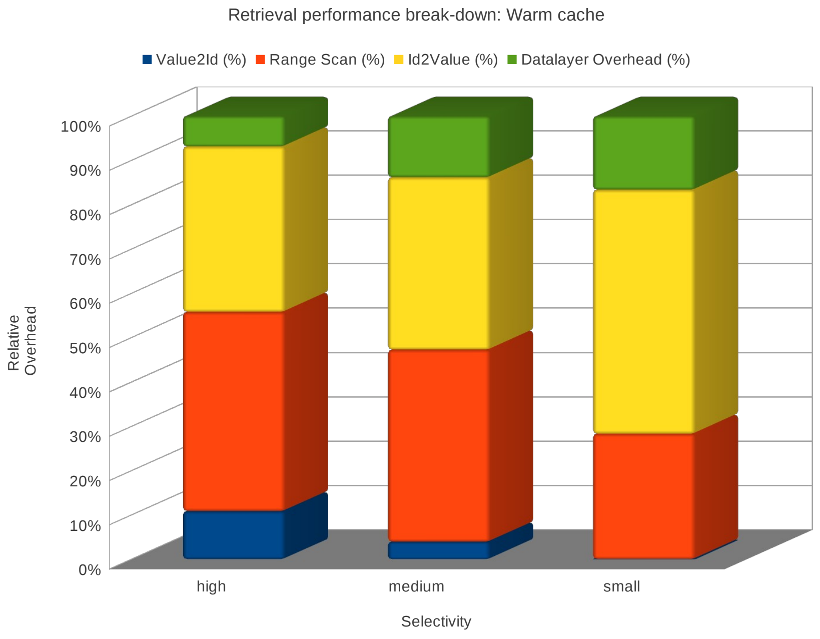


Figure 5.11: Retrieval Break-Down: Warm Cache

In figure 5.11, we show how the overhead distribution is affected by caching for the same queries. We now notice that the *Id2Value* and *RangeScan* take up almost equally sized fractions for high and medium selectivities. However, for small selectivity the *Id2Value* step makes up 26.8% more than the *RangeScan*, showing that the range scan is still more scalable. Still, this is a worse case scenario because, for this particular dataset, each quad element in our result set is unique. This means that for a simple query pattern of type `(?,P,?,?)` with 1 million statement results, we made a request for 3 million elements in the *Id2Value* step. In other datasets, many elements are reused, especially the graph name.

5.3.3 HDFS locality impact

As mentioned in section 4, there is a difference in data locality between the 2 loading techniques. We study here the impact that data locality has in retrieval performance. We note that, in time, the locality issue dissapears through HBase's *compactions*, because, when the internal files are rewritten, the HDFS Datanode preferred will be the one co-located with the *RegionServer* [45]. Still, the full data locality can be achieved only after a *MajorCompaction*, which is usually a lengthy process which takes up resources. Production systems usually schedule these processes at moments of minimal load in order not to affect Service Level Agreements (SLAs). Thus, after Map-Reduce bulk loading the system might run for a while without good data locality. Furthermore in a Region Server failover situation another Region Server may be assigned regions with non-local StoreFiles. All these motivate this section's measurements.

Each Region Server provides a measure for data locality named *hdfsBlocksLocalityIndex*. We retrieve this measure from each server and present the mean and standard deviation for all 13 region servers in the table below. The Coprocessors version does not have 100% locality because we load only the index tables through coprocessors, while the dictionary tables are loaded through Map-Reduce. However, their lack of data locality does not affect read performance too much. The *Value2Id* step has showed too little overhead to become a concern. And for the *Id2Value* table our intention is to fully cache it, so access to HDFS is minimized. The data locality for the M/R version is much worse than the Coprocessors version and its *hdfsBlocksLocalityIndex* varies between 18.3 and 23.3.

Table 5.2: Cluster parameters influencing read throughput

Dataset size	**hdfsBlocksLocalityIndex**		**numberOfStorefiles**	
	MR	**Coprocessors**	**MR**	**Coprocessors**
11.48 mill	18.3 (18.05)	86 (4.71)	8 (0.4)	10.571 (1.72)
113.41 mill	23.3 (9.34)	86.214 (3.57)	19.86 (3.11)	40.86 (8.0)
1.133 bill	20.38 (9.76)	84.21 (3.5)	40.3 (3.66)	112.31 (21.66)
DBPedia	23.3 (11.04)	85.93 (12)	35 (3.72)	54.14 (6.07)

We show the impact of data locality on read performace in figure 5.12. For this particular case study we used the same caching configuration for both M/R and Coprocessors - 40% memstore, 40% block cache. We ran our ad-hoc benchmark for the BSBM dataset of 113,413,428 quads in 3 cases:

1. immediately after doing a Map-Reduce bulk load

2. after doing a Major Compaction on 1)

3. after doing a bulk load using coprocessors

We present logarithmic trend lines for each case in figure 5.12. We notice the best trend line is exhibited for case (2).

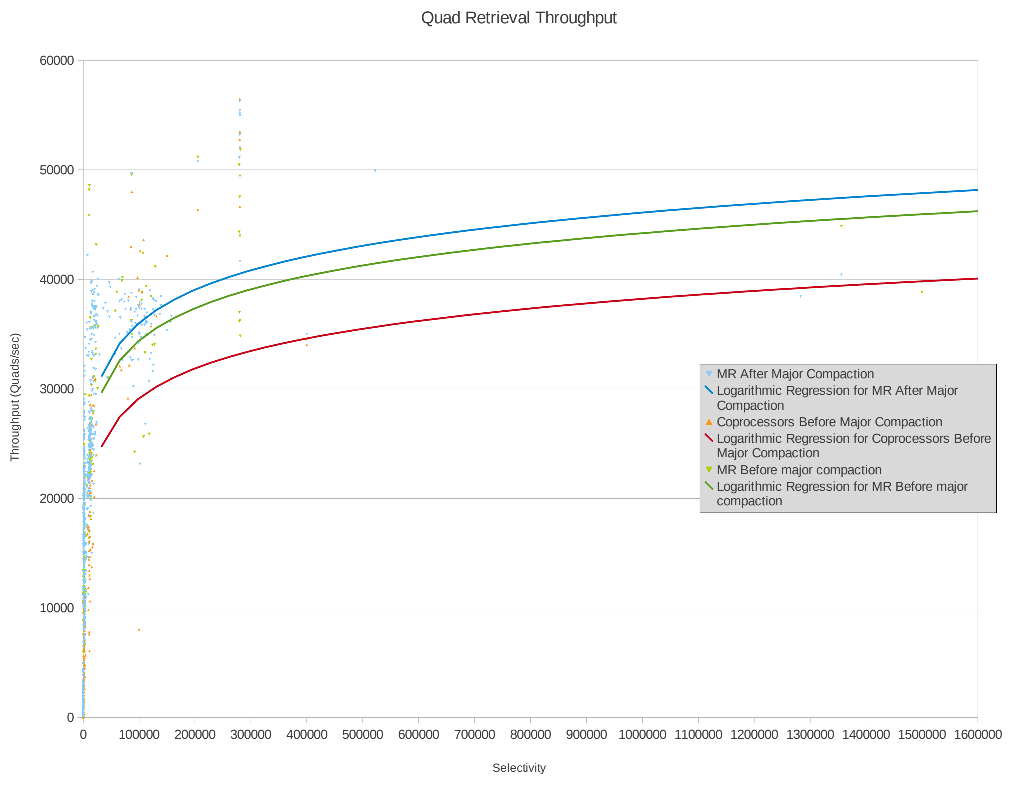


Figure 5.12: Quad Retrieval Throughput - 113 million quad BSBM dataset

Although (3) has very good data locality as shown in the table above, it performs worse even than (1). The reason is the high number of StoreFiles created by using the online API (show in table 5.2). As specified in section 2.3.2, when a read request is issued, HBase internally performs a *merged read* from a couple of StoreFiles, so if their number is too high, read performance will undoubtedly take a hit, even with Bloom Filters enabled. This is confirmed by the fact that version (2) performs the best, as the Major Compaction reduces the number of StoreFiles to 1 per region.

5.3.4 Overall caching impact

In this section, we show the impact that increased memory space for the block cache has on overall retrieval throughput. We consider the DBPedia dataset and run the same ad-hoc benchmark after the data has been loaded with the 2 loading techniques. We also do a *Major Compaction* in both cases to eliminate the overhead that might have come from lack of data locality or increased number of *Storefiles*.

For each loading technique, we have different caching configurations. The *RegionServer*'s memory is divided into space allocated for the *Memstore*, used for writing into HBase, and space allocated for the block cache, used for reading. The Coprocessors use HBase's API for writing, so we also need enough space in memory for the Memstore. For fairness reasons, we allocated 40% memstore and 40% block cache. Conversely, the Map-reduce version generates the files for HBase outside the *RegionServer* process, so we do not need to allocate as much *Memstore* space, leaving more room for the block cache. We allocated 10% memstore and 70% for block cache. In both cases we leave 20% of the process memory for internal use.

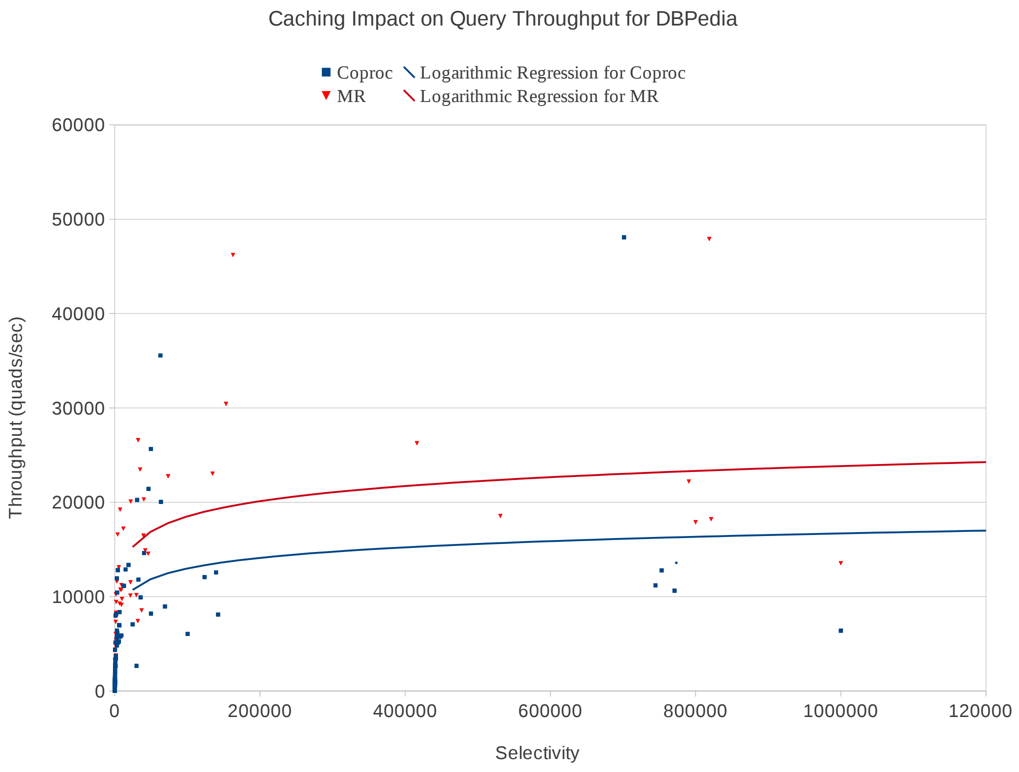


Figure 5.13: Caching impact on Query Throughput for DBPedia

In figure 5.13 we see that the difference in cache gives ˜30-40% increase in the overall retrieval throughput. We point out here that the 2GB RAM we allocated for the *RegionServer* is not enough for large datasets. HBase has a reputation of being memory hungry and typical production systems allocate 24 or even 48GB for each *RegionServer*.

5.3.5 Numerical data impact

As explained in section 3.3, numerical literals are not mapped in the dictionary tables. Instead, their internal Java representation is used directly in the index tables.

This effectively removes the *Id2Value* step in processing these elements. We show in this section the impact of this on retrieval performance.

We ran the ad-hoc benchmark 2 times on the DBPedia dataset. In the first run we use main queries that are known to retrieve only numerical literals in the O position i.e. `(?,<http://www.w3.org/2003/01/geo/wgs84_pos#lat>,?,?)`. Conversely, in the second run, we use main queries which do not return any numericals i.e. `(?,<http://xmlns.com/foaf/0.1/name>,?,?)`.

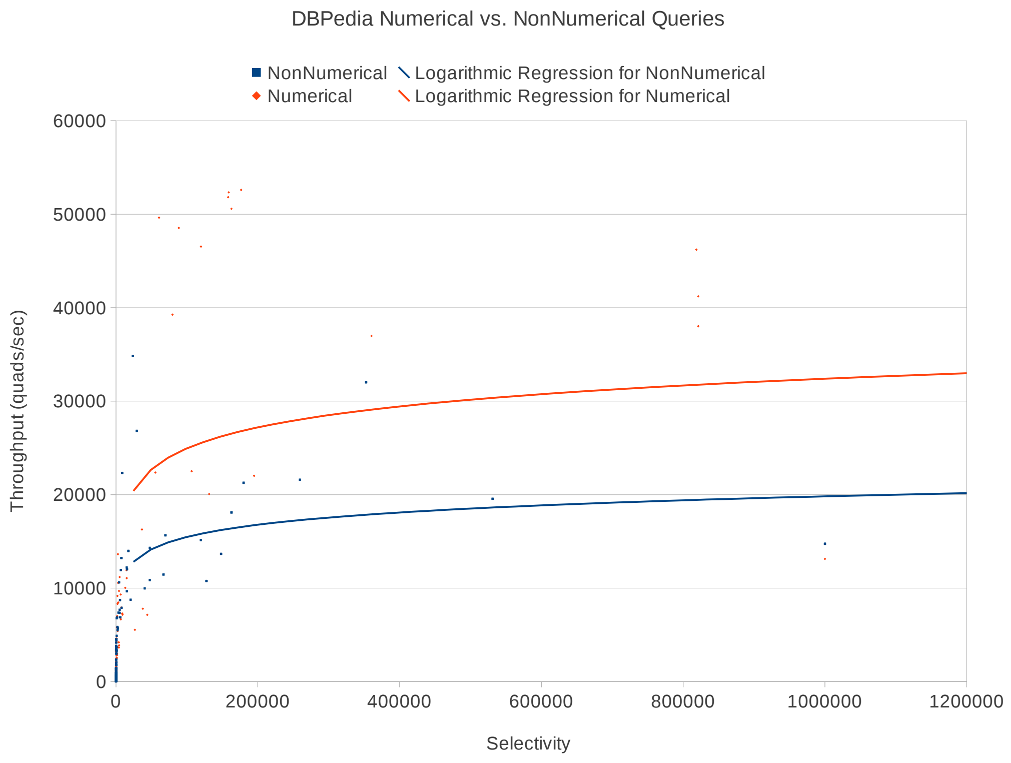


Figure 5.14: Numerical data retrieval impact on DBPedia

In figure 5.14 we notice that the version containing numericals has an increased average throughput of ˜40-50%. The peak was 52597 quads/second obtained for a selectivity of 3356.

5.4 Disk usage

For each dataset, we measured the schema's total disk usage. In table 5.3, we compare it with the disk usage of the input dataset. These are values reported by HDFS, so the total disk usage (column *Actual*) is obtained by multiplying with the replication factor in-use, in our case 3.

We notice that the index to input ratio stays at ~100% for the BSBM dataset but for DBPedia it reaches 157%. We point out that this is also explained by the difference in number of unique ids between the 2 datasets. Thus the tables *Value2Id* and *Id2Value* make up a fraction of 10%, respectively 9.3% from the input size of the DBPedia dataset. Meanwhile the same tables make up 1.33% and 5.98% of the 1.11 billion BSBM dataset.

Table 5.3: Disk Usage

Dataset Size	**Input (GB)**	**Index (GB)**	**Actual (GB)**
11.48 mill	3.940	4.140	12.419
113.41 mill	39.320	41.119	123.358
1.133 bill	390.200	387.501	1,162.503
385.84 mill (DBPedia)	84.020	132.350	397.049

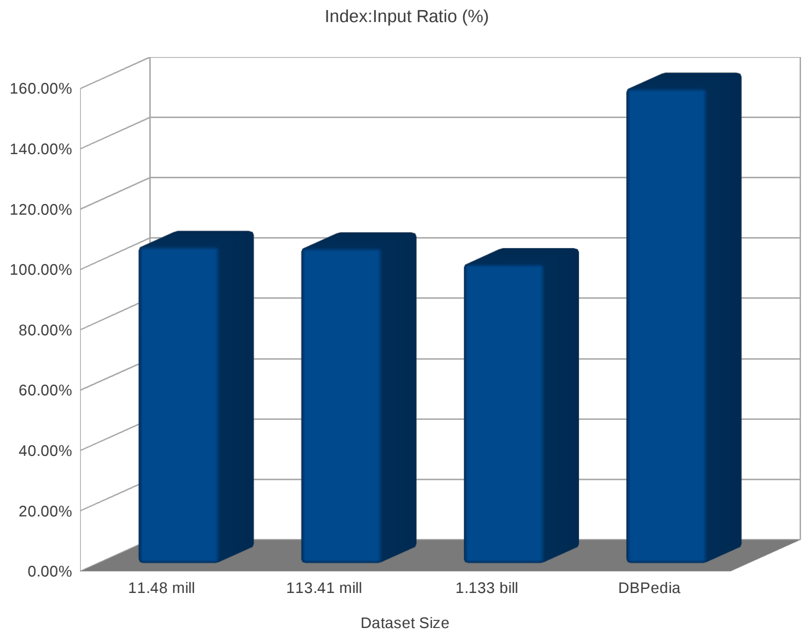


Figure 5.15: Disk Usage

We also point out that HBase stores additional metadata in a physical Key Value pair [10]. For our case in which the row key is 33 bytes and the column family name takes up 1 byte, the physical key value pair has 54 bytes. That translates to approximately 63% additional data that we do not actually need. However, this meta data makes the format self-contained and allows HBase to easily move data around internally. Methods for reducing disk usage include compression and block-encoding

[8] which has been recently introduced in version 0.94.0.

In this chapter we evaluated the Id-based schema on bulk loading and retrieval performance. For bulk loading, the M/R technique showed improved scalability with input dataset size, while the Coprocessors version exhibited poor performance for large datasets due to overhead from HBase's internal operations. However, the M/R technique also needs further improvements in terms of load balancing in order to achieve its full scalability potential. For retrieval, we showed that read throughput takes a significant hit if the *Id2Value* table can not be fully cached. Further analysis showed the impact that 3 factors have on read troughput: HDFS data locality, number of *Storefiles* and avalable block cache. We conclude that the M/R technique leaves the system in a state which offers better read throughput than the Coprocessors version. However, we point out that, in both cases, the maximum retrieval performance is achieved after doing a *Major Compaction* on the whole system. Finally, we evaluated the disk usage of our schema and obtained approximately a 1:1 ratio between the input BSBM dataset and the HBase tables.

Chapter 6

Related Work

In this section, we consider other triple stores that have been implemented and compare them with our implementation from a schema design point of view and, where possible, in terms of performance.

Schema Design Comparison

Historically, triple stores have emerged on top of Relational Databases e.g. Jena[15], Sesame[37]. The first schemas were either simply storing all triples into one table or they were property centric. The latter grouped subject-object pairs either by the type property of subjects or by clusters of properties [34].

Weiss et al. developed the Hexastore [59] schema with the goal of breaking away from the schemas restrained by the assumption of a pre-determined number of predicates [34] [60]. We agree with this direction especially when a significant large scale is required for a triple store, which implies a broad diversity of RDF data. Furthermore, having a table for each predicate as suggested in [34] implies a lot of joins when querying for all the properties associated to a certain subject.

The Hexastore [59] schema is designed for storing only triples and consists of 6 indexes that cover all possible combinations of s,p,o. Their solution however is for an in-memory single machine store, so they do not address scalability. Instead, Jianling Sun[53] has adopted it for HBase by using a separate table for each of the 6 indexes. His approach is to store the verbose RDF data into the row key and column qualifier e.g. for the P_SO table a predicate is stored as row key and all its associated subject-object pairs will be stored as separate column qualifier names. One disadvatange of this schema pointed out by the author himself is the 6x additional storage space required, which is also multiplied by the HDFS replication factor. Furthermore we point out that this design leads to a flat-wide table, which as shown in section 3.4 does not favor Region splitting and in turn fine-grained load balance is affected. Finally, this design can not be extrapolated when using quads because it explodes from 3!=6 to 4!=24 tables. Weiss et al. [59] also admits that their design is possible due “to the very triple nature of RDF data”.

[59] and [50] point out disadvantages to the schema employed in [46] and [61], which is also the schema we used in our system. They suggest that it is only well-suited for simple *statement-based* queries, because it does not cover all the possible combinations of s,p,o,c elements e.g. none of the tables provides a sorted list of subjects for a given predicate. Their argument is rooted in the fact that joins have to be reduced to fast merge-joins as much as possible. However they do not provide a solution for the case of quads without using all 24 combinations. The alternative to merge-joins are nested-loop joins which are expensive also in the context of HBase. They imply a lot of random reads, which, based on our evaluation results, should be avoided as much as possible. We show possible solutions to these problems in chapter 7.

RDF3X[50] is currently the fastest triple store designed for a single machine. It is based on clustered B+ trees, so inherently it can not scale past a certain point. More specifically, range-scans in B+ trees do not guarantee that data will be read sequentially from disk. HBase was specifically designed to avoid the scalability issues of B+ trees. In fact, RDF3X manages updates by loading data first into a differential index which is then merged with the core index. We point out that this operation is very similar to HBase’s internal *compactions*, although their differential index is stored in memory. The index is based on all 6 combination of s,p,o and uses a compressed scheme in which only differences between keys are stored. This feature has also been introduced in HBase starting with version 0.94.0 under the term *block encoding*[8]. The similarity to our scheme consists in associating triple elements to ids using 2 dictionary tables. The difference is that they use a direct-mapping [42] index for mapping ids back to strings which is expected to perform better than the sparse index that HBase provides. For joins, they rely on statistics to build a query plan using dynamic programming.

Papailiou et. al [51] developed H2RDF, a distributed triple-store also based on HBase. They convert the verbose RDF data to 8-byte MD5 hashes which are combined into 3 index tables pos,spo and osp to cover all possible access patterns. We point out that this solution can not scale past a certain point because of hash collisions i.e. for 2 billion unique RDF elements there is a 1 in 10 chance of collision. They store the first 2 elements of the index in the row key and the last one in the column qualifier. This does not lead to a very high number of columns so we consider it an equivalent alternative to our schema. Similar to our approach, they rely on prefix matching to quickly answer simple queries (without joins). For joins they rely on selectivity counters and statistics to implement a complex query planner.

Tsialiamanis et. al [54] have deployed a triple store on top of MonetDB, another column-oriented NoSQL database. They show a couple of heuristics for efficient query planning without using statistics which add overhead in terms of maintainability, storage space and query execution. They also use 6 indexes for every combination of s,p,o and design the planner with the main goal of maximizing merge-joins. With the exception of large star-joins, they show improved results over RDF3X.

Another important contribution is made by Khadilkar et. al [57]. They develop a triple store that combines the Jena framework [15] with the storage provided by HBase. They evaluate their store with 6 different storage layouts. The one similar to our design is the Indexed layout, and consists of 6 index tables for all combinations of s,p,o, which are stored in the row key in verbose format. This layout is the one that performs the best when loading, but it is also the one that takes up the most disk space and the one with the worst response time. However, we point out that the authors implemented a naive nested-loop approach for handling joins, which does not leverage having all combinations of s,p,o to implement merge-joins. In these conditions, the query results are expected to be so bad because of the repeated random reads. The layout with the best response time is a Hybrid layout which combines vertical partitioning with 2 additional tables s-po and o-sp. However, this layout, as well as the others, are replicated for each separate graph, which can lead to an explosion of the number of tables. This in turn increases the number of regions per RegionServer, which is recommended to stay in the hundreds range for HBase to maintain good performance.

YARS2[47] is a federated RDF repository which uses our indexing strategy. For implementing the index, they choose a *sparse index* - employed also by HBase - over B-trees and Hashtables. They also show that smaller block sizes favor faster random access within a table. Their implementation of joins, based on index nested loop joins, shows decent performance for 1 and 2 joins, maintaining a response time below 1 second for most queries. However, index nested loop joins are known to perform good for selective queries and they do not show results for more than 2 joins and result sets above 300k quads.

Recently a new format - HDT [43] - for publishing and exchanging RDF data has been standardized. This format consists of a dictionary encoding the verbose data to a set of ids and a facts table which contains the graph structure represented using these ids. Importantly, this structure maps very well on our schema and it can remove altogether the id-generation stage from our bulk loading process, which, as we showed, can take up to 43% of total loading time.

Evaluation Comparison

Another HBase based triple store was developed by Choi et. al[39]. They do not present the schema they used in their implementation, so a comparison can not be made in that aspect. For evaluation, they also use the BSBM dataset and claim 10k triples/second for loading 100 million triples on 6 nodes, so 1666 triples/sec/node. Our loading performance showed better results: 32152 quads/second for the 113 million quads dataset on 13 nodes, so 2473 quads/sec/node.

RDF3X[50] presents an evaluation on 3 datasets - Barton, Yago and Librarything. They exhibit a loading throughput that varies between 1.36MB/s and 3.67MB/s on one node. However they mention this is after bringing the triples into a "factorized form": fact tables and dictionary. So they do not present the overhead for generating ids, which from our evaluation can take up an important fraction. We compute our equivalent throughput for this second stage and get between 14.4MB/s and 21MB/s on 13 nodes, depending on dataset size, for the BSBM dataset.

H2RDF[51] loads 1.3 billion triples in 118 minutes using 9 nodes and 10 map-reduce slots per node, resulting in a loading throughput of 2040 triples/second/slot. We load 1.133 billion quads in 693 minutes using 13 nodes with 4 map-reduce slots per node, resulting in 524 quads/second/slot. We point out that they do not need an id-generation stage because of hashing and that they consider triples without metadata so they need to load only 3 index tables instead of 6. Their query evaluation on LUBM shows better results than a pure Hadoop triple store [48] but can only beat RDF3X[50] for a single query.

In this chapter, we compared our work with other triple/quad stores and made some interesting insights. Notably, we saw that our schema was successfully employed in other projects as well, but that it exhibits a significant drawback when handling joins. From a loading perspective, it showed promising results in other work as well, so we are confident it was a good choice in that regard.

Chapter 7

Future work

In this chapter, we present ideas for future work on this project. We divide the discussion between loading and retrieval improvements.

Retrieval improvements

As we saw in chapter 6, our schema needs to be improved in order to handle joins. We see multiple possible solutions:

- Add more index tables. We need to find a trade-off between the number of index tables generated and the overall query performance. Our evaluation of disk usage showed good results, so we can afford to add other index tables for efficiently solving more frequent join patterns.

- Use coprocessors to reorder results so that merge-joins are favored on the client. For example, if we want to do a subject-object join i.e. `(?x,p1,?y,c1)` with `(?y,p2,?z,c1)`, they will both go to the CPSO table, so the results are ordered by subject. We can use coprocessors to reorder the results from the first query by object, so the client can now make a fast merge join. This would have an impact only if the results set is big enough to be processed on multiple Region Servers.

- Use coprocessors to do the joins entirely without including the client in the loop: for example if 2 query patterns need to be joined, we can use coprocessors to write the intermediate results of the first query on HDFS, read them back when intercepting the results of the second query and do the join within each coprocessor.

Another improvement can be made in the *Id2Value* step. In section 5.3.2 we saw that, ideally, the *Id2Value* table should have to fit into memory. However if the hardware does not allow that, one optimization would be to split the *Id2Value* table into 2 *column families*: one that stores URIs, blank nodes and short literals e.g. below 1024 bytes and another one that stores only long literals. The idea is that long literals might take up too much space in the block cache, which could have been used by many other short elements. HBase has the nice feature of allowing different caching configurations per *column family*, so we enable it only for the first *column family*, while keeping the second one on disk. To distinguish between the 2 categories of ids, we can use the first bit and encode 0 prefixed ids to short elements, while 1 prefixed ids corespond to long literals.

Another direction for improving retrieval is to further optimize the HBase client. Here we have 2 options:

- Cache ontology elements on the client: given the reduced size of ontologies and their frequent use, we can cache the mapping of URIs that make up these ontologies on the client side to further improve the *Id2Value* step.
- Overlap the RangeScan with the Id2Value step: either implement a multi-threaded synchronous client which starts the random gets as soon as the range scan finishes the first *next* call; either use the asynchronous, non-blocking and thread-safe client developed by StumbleUpon [1] to hide latency

An alternative solution that might work better when retrieving large result sets is to use coprocessors to make the *Id2Value* mapping in a distributed fashion. We can use observer coprocessors to intercept the range scan *next* calls, map the ids back to strings within our deployed user function and then transfer to the client the final string-based quads.

In order to properly evaluate query performance and compare it with other triple stores, we need to implement support for SPARQL. Work has already started for this direction and the intention is to use Sesame2 [28] as a frontend which parses SPARQL queries, and a custom implementation of the Storage And Inference Layer (SAIL) API as the backend which acts as the HBase client.

Loading improvements

On the loading side the most important improvement that still needs to be made is load balancing. To take into account data skew, we can build histograms during the id generation stage which tell us how the data is distributed across id ranges. We then use these histograms to properly choose table splits.

The load balancing problem becomes more complex when we take into account incremental bulk loads. Silberstein et al. [52] addresses the problem by introducing a planning stage before the actual bulk load. The planning phase has to take into account the tradeoff between the cost of splitting and moving partitions before the bulk load and the gain that would be obtained in throughput. They formalize the problem by reducing it to a problem of packing vectors [52].

Chapter 8

Conclusions

In this project, we adapted an Id-based schema in the context of HBase and showed that it enables good loading performance through a compact representation of RDF data. The schema also enables good retrieval performance, because we saw that the step which represents the bulk of the entire operation - the Range Scan - is not the main bottleneck. We also showed how numerical range-predicates can be handled efficiently by translating them into Range Scans.

The retrieval performance could not be compared with other triple stores because we did not implement functionality for SPARQL. However, we designed our own benchmark and measured a peak read throughput of 56447 quads/second.

For bulk loading, we evaluated 2 techniques, one based on Map-Reduce, as suggested by the HBase guide [32], and a novel one based on HBase coprocessors. The M/R technique showed improved scalability with input dataset size, while the Coprocessors version became affected for large datasets due to overhead from HBase's internal operations. Still, for the M/R technique, there are further improvements to be made in terms of load balancing in order to achieve its full scalability potential.

Additionally, the M/R technique leaves the system in a state which provides better retrieval performance. Although it does not achieve good HDFS data locality, the reduced number of *Storefiles* and the increased block cache available favors improved read throughput.

In conclusion, the M/R loading technique proves to be the best choice in combination with our Id-based schema. Still, consistent work needs to be done both on the loading and the retrieval side to reach a full fledged quad store. We suggested solutions for load balancing and handling joins in order to get closer to this goal.

Appendix A

Evaluation Tables

Table A.1: M/R Loading Throughput Break-down (Quads/sec)

Dataset size	IdGen+SPOC	Secondary Indexes	Dictionary
11.48 mill	28,503.360	139,553.949	47,599.940
113.41 mill	64,980.261	393,303.357	191,340.595
1.133 bill	66,189.799	260,401.870	231,037.527

Table A.2: M/R Bulk Loading Break-down

Dataset size	IdGen+SPOC	Secondary Index Tables	Dictionary Tables
11.48 mill	41.398%	42.277%	16.325%
113.41 mill	49.480%	40.874%	9.646%
1.133 bill	41.156%	52.306%	6.537%
385.84 mill (DBPedia)	40.779%	51.422%	7.799%

Table A.3: M/R Index Tables Loading Throughput Break-down (Quads/sec)

Dataset size	SPOC	CSPO	CPSO	OCSP	OSPC	POCS
11.48 mill	141,745.19	138,418.43	139,958.78	136,045.42	152,303.61	132,578.75
113.41 mill	516,946.58	492,369.73	508,817.69	334,707.90	352,667.95	345,627.23
1.133 bill	346,152.41	226,656.98	295,298.64	264,321.76	237,787.45	292,908.86

Table A.4: Coprocessors Loading Throughput Break-down (Quads/sec)

Dataset size	IdGen+SPOC	SecondaryIndex Tables	Dictionary Tables
11.48 mill	29,562.103	247,230.696	47,599.940
113.41 mill	66,089.238	337,933.818	191,340.595
1.133 bill	59,210.579	166,661.066	231,037.527

Table A.5: Coprocessors Loading Break-down

Dataset size	IdGen+SPOC	Secondary Index Tables	Dictionary Tables
11.48 mill	44.029%	26.324%	18.007%
113.41 mill	44.928%	43.933%	8.908%
1.133 bill	34.165%	60.690%	4.854%
385.84 mill (DBPedia)	31.673%	56.114%	5.676%

References

[1] Asynchronous hbase. https://github.com/stumbleupon/asynchbase. 7

[2] Cassandra nosql database. http://cassandra.apache.org. 1.1

[3] Couchdb nosql database. http://couchdb.apache.org/. 1.1

[4] Ganglia monitoring system. http://ganglia.sourceforge.net. 5.1

[5] Hadoop and mapreduce: A parallel program to assign row numbers. http://blog.data-miners.com/2009/11/hadoop-and-mapreduce-parallel-program.html. 4.1

[6] Hadoop filesystem (hdfs). http://hadoop.apache.org/hdfs. 1.1

[7] Hadoop task side-effect files. http://hadoop.apache.org/common/docs/r1.0.0/mapred_tutorial.htm
Effect+Files. 4.1

[8] Hbase block encoding. https://issues.apache.org/jira/browse/HBASE-4218. 5.4, 6

[9] Hbase coprocessors introduction. https://blogs.apache.org/hbase
/entry/coprocessor_introduction. 1.2, 4

[10] Hbase keyvalue. http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/KeyValue.html. 5.4

[11] Hbase m/r bulkload. http://hbase.apache.org/book/arch.bulk.load.html. 1.2, 4.1

[12] Hbase number of column families. http://hbase.apache.org/book/number.of.cfs.html. 2.3.1

[13] Hbase schema design - things you need to know. http://www.youtube.com/watch?v=_HLoH_PgrLk&feature=related. 3.4

[14] Is the relational database doomed? http://www.readwriteweb.com/enterprise/2009/02/is-the-relational-database-doomed.php. 1.1

[15] Jena framework. http://jena.apache.org. 6

[16] Linking open data. http://www.w3.org/wiki/SweoIG/TaskForces /CommunityProjects/LinkingOpenData. 1.1

[17] Neo4j - a graph database. http://highscalability.com/neo4j-graph-database-kicks-buttox. 1.1

[18] Neo4j nosql database. http://neo4j.org. 1.1

[19] No sql guide. http://nosql-database.org. 1.1

[20] Nosql databases on amazon web services. http://aws.amazon.com/nosql. 1.1

[21] Open phacts project. http://www.openphacts.org. 1.1

[22] Optimizing writes in hbase. http://gbif.blogspot.nl/2012/07/optimizing-writes-in-hbase.html. 4.2

[23] Owl web ontology language. http://www.w3.org/TR/owl-features. 2.2

[24] Rdf convert tool. http://sourceforge.net/projects/rdfconvert. 5.2

[25] Secondary indexing. http://wiki.apache.org/hadoop/Hbase/SecondaryIndexing. 4.2

[26] Semantic web. http://www.w3.org/2001/sw. 1.1

[27] Sesame value. http://www.openrdf.org/doc/sesame2/api/org/openrdf/model/Value.html. 3.3

[28] Sesame2. http://www.openrdf.org. 7

[29] The storage technologies behind facebook messages. http://www.youtube.com/watch?v=cSNGGAKJqwk&feature=g-vrec&context=G2e8c625RVAAAAAAAAAg. 1.1

[30] Voldemort nosql database. http://project-voldemort.com/http://project-voldemort.com. 1.1

[31] *Hadoop: The Definitive Guide*, chapter How MapReduce Works. O'Reilly Media, Inc., 2009. 4.1

[32] *HBase: The Definitive Guide*. O'Reilly Media, 2011. 1.1, 2.3, 2.3.2, 2.3.4, 8

[33] *Synthesis Lectures on the Semantic Web: Theory and Technology*, chapter Linked Data: Evolving the Web into a Global Data Space. Morgan & Claypool, 2011. 1.1, 2.1, 2.2

[34] Daniel J. Abadi, Adam Marcus, and Barton Data. Scalable semantic web data management using vertical partitioning. In *In VLDB*. 6

[35] Antonio Barbuzzi, Pietro Michiardi, Ernst Biersack, and Gennaro Boggia. Parallel bulk insertion for large-scale analytics applications. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware*. 4.1

[36] C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):122, 2009. 2.1

[37] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. Sesame: An architecture for storing and querying rdf data and schema information. In *Semantics for the WWW*. 6

[38] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *IN PROCEEDINGS OF THE 7TH CONFERENCE ON USENIX SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION - VOLUME 7*, pages 205–218, 2006. 1.1, 2.3

[39] Hyunsik Choi, Jihoon Son, YongHyun Cho, Min Kyoung Sung, and Yon Dohn Chung. Spider: a system for scalable, parallel / distributed evaluation of large-scale rdf data. In *Proceedings of the 18th ACM conference on Information and knowledge management*. 6

[40] W3C Dan Brickley. Rdf vocabulary description language 1.0: Rdf schema. http://www.w3.org/TR/rdf-schema. 2.2

[41] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 2008. 4

[42] Andr Eickler, Csrsten A. Gerlhof, and Donald Kossmannt. A performance evaluation of oid mapping techniques. In *In Proceedings of the 21st VLDB Conference*. 6

[43] Javier D. Fernández. Binary rdf for scalable publishing, exchanging and consumption in the web of data. In *Proceedings of the 21st international conference companion on World Wide Web*. 6

[44] The Apache Software Foundation. Zookeeper. http://zookeeper.apache.org/doc/current/zookeeperOver.html. 2.3.2

[45] Lars George. Hbase file locality in hdfs. http://www.larsgeorge.com/2010/05/hbase-file-locality-in-hdfs.html. 5.3.3

[46] Andreas Harth and Stefan Decker. Optimized Index Structures for Querying RDF from the Web. In *IEEE LA-WEB*, pages 71–80, 2005. 1.2, 1.3, 3.3, 6

[47] Andreas Harth, Jrgen Umbrich, Aidan Hogan, and Stefan Decker. Yars2: A federated repository for querying graph structured data from the web. In *of Lecture Notes in Computer Science*. 3.4, 6

[48] Mohammad Husain, James McGlothlin, Mohammad M. Masud, Latifur Khan, and Bhavani M. Thuraisingham. Heuristics-based query processing for large rdf graphs using cloud computing. *IEEE Trans. on Knowl. and Data Eng.*, 2011. 6

[49] Tim-Berners Lee. Linked data. http://www.w3.org/DesignIssues/LinkedData.html. 1.1

[50] Thomas Neumann and Gerhard Weikum. The rdf-3x engine for scalable management of rdf data. *The VLDB Journal*, 2010. 6, 6

[51] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, and Nectarios Koziris. H2rdf: adaptive query processing on rdf data in the cloud. In *WWW (Companion Volume)*. 6, 6

[52] Adam Silberstein, Brian F. Cooper, Utkarsh Srivastava, Erik Vee, Ramana Yerneni, and Raghu Ramakrishnan. Efficient bulk insertion into a distributed ordered table. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 7

[53] Jianling Sun. Scalable rdf store based on hbase and mapreduce. In *Advanced Computer Theory and Engineering (ICACTE), 2010 3rd International Conference*. 6

[54] Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides, and Peter Boncz. Heuristics-based query optimisation for sparql. In *Proceedings of the 15th International Conference on Extending Database Technology*. 5.3, 6

[55] Jacopo Urbani. Rdfs/owl reasoning using the mapreduce framework. Master's thesis, Vrije Universiteit Amsterdam, 2009. 4.1, 4.1

[56] Jacopo Urbani, Spyros Kotoulas, Jason Maassen, Niels Drost, Frank Seinstra, Frank Van Harmelen, and Henri Bal. H.: Webpie: A web-scale parallel inference engine. In *In: Third IEEE International Scalable Computing Challenge (SCALE2010), held in conjunction with the 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid*, 2010. 1.1

[57] Bhavani Thuraisingham Vaibhav Khadilkar, Murat Kantarcioglu. Jena-hbase: A distributed, scalable and efficient rdf triple store. Technical report, The University of Texas at Dallas, 2011. 6

[58] W3C. Skos simple knowledge organization system. http://www.w3.org/TR/2009/REC-skos-reference-20090818. 2.2

[59] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proc. VLDB Endow.*, 2008. 6

[60] Kevin Wilkinson. Jena property table implementation. In *SSWS*. 6

[61] David Wood. Kowari: A platform for semantic web storage and analysis. In *In XTech 2005 Conference*. 3.3, 6